6.858: Computer Systems Security

Fall **2012**

Home

General Information

Schedule

Reference Materials

Piazza discussion

2011 Class Materials

General Information

Catalog description // Who should take 6.858 // Communication // Grading // Turn-in // Collaboration // Class meetings // Staff // TA office hours

MIT catalog description

Prereq.: 6.033
G (H)
3-0-9
4 EDP

Design and implementation of secure computer systems. Lectures cover threat models, attacks that compromise security, and techniques for achieving security, based on recent research papers. Topics include operating system (OS) security, capabilities, information flow control, language security, network protocols, hardware security, and security in web applications. Assignments include labs that involve implementing and compromising a secure web server and web application, and a group final project.

Students can use 6.858 to fulfill the engineering concentration requirements for Computer Systems.

Who should take 6.858?

6.858 is primarily intended for seniors and M.Eng students who want to learn about how to build secure computer systems in detail. PhD students are also welcome; 6.858 counts as a systems TQE subject.

Communication

We will distribute assignments and announcements on the course web site. We expect students to check the 6.858 home page for both news and assignments at least once a week. If you hear a rumor, check it there.

Grading policy

Grades in 6.858 will be based on the results of two quizzes (one in the middle of the term and one in the next-to-last week of classes, 20% in total), lab exercises (35%), final project and presentation (25%), and class participation and homeworks (together 20%). No

quiz during final exam week.

Turn-in policy

You are required to turn in each lab; if you have not turned in all of the labs, you will receive an F. Labs that are turned in but score 0 points will receive a D. You have a total of 3 late days to use throughout the semester. There are no partial late days: an assignment that is only six hours late uses an entire late day. After you have used up your late days, each additional day late will incur a full letter grade penalty. Saturday and Sunday both count as days.

Collaboration

You may not collaborate on quizzes. You are welcome to discuss the labs with other students, but you should complete all assignments on your own, and you should carefully acknowledge all contributions of ideas by others, whether from classmates or from sources you have read. Final projects will be in groups, where you should collaborate.

Class meetings

Lectures will be held MW 11-12:30 in 32-144.

Staff

Lectures

Nickolai Zeldovich	32-G994	x3-6005	nickolai@csail.mit.edu

Teaching assistant

David Benjamin	TBD	davidben@mit.edu

Course mailing list: *6.858-staff@pdos.csail.mit.edu*
Use this mailing list to contact all the 6.858 staff.

TA office hours

Office hours will be held TBD. If you can't make it, you can email us to set up another time to meet. You are welcome to stay and ask questions after lectures.

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Friday, 18-May-2012 13:52:46 EDT*

6.858: Computer Systems Security

Fall **2012**

Home

General Information

Schedule

Reference Materials

Piazza discussion

2011 Class Materials

Reading materials

Cryptography

- *Applied Cryptography* by Bruce Schneier. John Wiley & Sons, 1996. ISBN 0-471-11709-9.
- Handbook of Applied Cryptography by Menezes, van Oorschot, and Vanstone.
- *Introduction to Cryptography* by Johannes Buchmann. Springer, 2004. ISBN 978-0-387-21156-5.
- Cryptographic libraries:
  - KeyCzar by Google.
  - GPGME by GnuPG.
  - OpenSSL.
  - NaCl: Networking and Cryptography library by Tanja Lange and Daniel J. Bernstein.

Control hijacking attacks

- Smashing The Stack For Fun And Profit, Aleph One.
- Bypassing non-executable-stack during exploitation using return-to-libc by c0ntex.
- Basic Integer Overflows, blexim.
- *The C programming language (second edition)* by Kernighan and Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8, 1998.
- Intel 80386 Programmer's Reference Manual, 1987. Alternatively, in PDF format. Much shorter than the full current Intel architecture manuals below, but often sufficient.
- IA-32 Intel Architecture Software Developer's Manuals, Intel, 2009. Local copies:
  - Volume 1: Basic Architecture
  - Volume 2A: Instruction Set Reference, A-M
  - Volume 2B: Instruction Set Reference, N-Z
  - Volume 3A: System Programming Guide, Part 1
  - Volume 3B: System Programming Guide, Part 2

Web security

- Browser Security Handbook, Michael Zalewski, Google.
- Browser attack vectors.
- Google Caja (capabilities for Javascript).
- Google Native Client allows web applications to safely run x86 code in browsers.
- Myspace.com - Intricate Script Injection Vulnerability, Justin Lavoie, 2006.

- The Security Architecture of the Chromium Browser by Adam Barth, Collin Jackson, Charles Reis, and the Google Chrome Team.
- Why Phishing Works by Rachna Dhamija, J. D. Tygar, and Marti Hearst.

OS security

- Secure Programming for Linux and Unix HOWTO, David Wheeler.
- setuid demystified by Hao Chen, David Wagner, and Drew Dean.
- Some thoughts on security after ten years of qmail 1.0 by Daniel J. Bernstein.
- Wedge: Splitting Applications into Reduced-Privilege Compartments by Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp.
- KeyKOS source code.

Exploiting hardware bugs

- Bug Attacks on RSA, by Eli Biham, Yaniv Carmeli, and Adi Shamir.
- Using Memory Errors to Attack a Virtual Machine by Sudhakar Govindavajhala and Andrew Appel.

Mobile devices

- iOS security

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Thursday, 23-Aug-2012 21:19:26 EDT*

6.858 Day 0

9/5

6.324 recommended to me personally by prof — Systems

Nickolai Zeldovich

Building Secure Systems

Lectures M, W

No recitation

Thur — lab 1 tutorial basic info
- C
- GDB

Piazza

Lectures: Discuss research papers

Not really Crypto (6.857)

Hard to find actual secure systems

②

Class lab heavy
- ~~Build~~ hack
  - build secure systems
  - every lab a new lang
  - 2-3 people final project
    - can combine w/ other classes

Disclaimer from the lawyers

Today: High levels examples
- Goal
- Presence of adversary   ie access email w/o others accessing

Lots of ways to monotize

1. Policy → goals
   - Confidentiality } data
   - integrity } data
   - availability } live-news

③

Threat Model - who is bad guy
assumptions what he can/can't do
be generous

Mechanisms - What enforce policy

Goal is negitive -> the absence of anything going wrong

Imperfect Security: Tradeoff b/w functionality + risk
Better mechanisms enable better security

Where does security go wrong?
All 3

(4)

Policy gone wrong — HS system in VA

- students — submit assignment
- teacher — view all files in class + add students
- superintendent — read everything

But if student gets password for 1 teacher
- can add superintendent to class
- now teacher can read everything!

What went wrong: No clear goal of what should not be allowed

Yahoo Mail — asks username + password
but can do security qn if forgot
often public info → what HS?

Amazon - Apple — 3 phone calls
- Wired editor

(5)

Broad goals

← Bad assumptions

1. ie Users pick good passwords
   User replies to phishing
   User falls for social engineering
   "rubber hose crypto analysis"

2. Bad SSL certs
   mistrusted root authority
   few 100 in modern browsers
   assume they will do the ~~wrong~~ right thing always
   DigiNotar

3. Random #s can be generated
   embedded devices actually bad at
   virtual machines

(6)

have same random seeds

"Orange Book" - Security OS design from gov

"Red Teams" - hackers hired to attack

Source Code repo not secure

Bad guy in some part of networks
but not others

ie firewall
└ bad guy only outside firewall

like NYT ad loads img
http:// 192.168.1.1/?D= firewall-disab

Not on internet is fine
Till update it w/ a USB drive

Rember Tradeoff
how parinoid are you?

7

Mechanisms

More tech side

Bugs in implementation of System

Very hard to write bug free software

Buffer Overflow

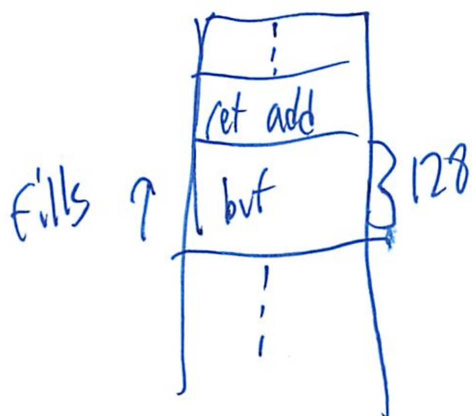
Many things written in C

Not type- or memory- safe

```
int read_num(void) {
    char buf[128]
    gets(buf)
    return atoi(buf);
}
```

⑧

Runs on stack



gets() does not know how big buf is

So can easily overwrite return address

w/ 128th to 131th byte is special

returns to special area

which does something special

Almost every mistake can lead to a security issue

- integer overflow

int32 x, y

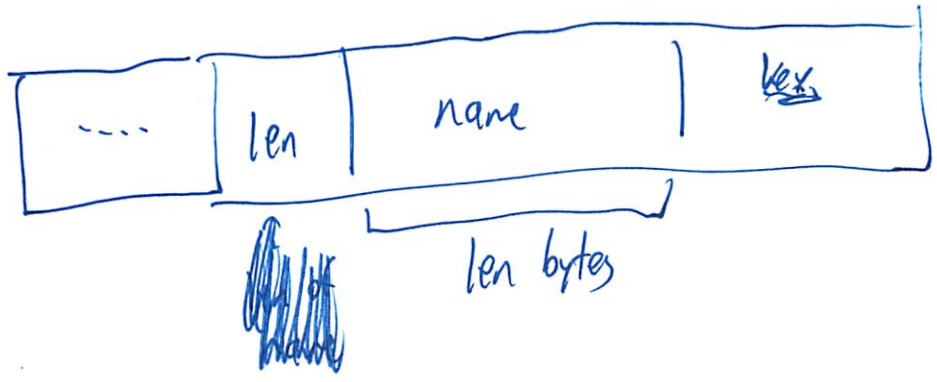
x · y

↑ can wrap around in 32 bits

* must worry about every corner case

⑨

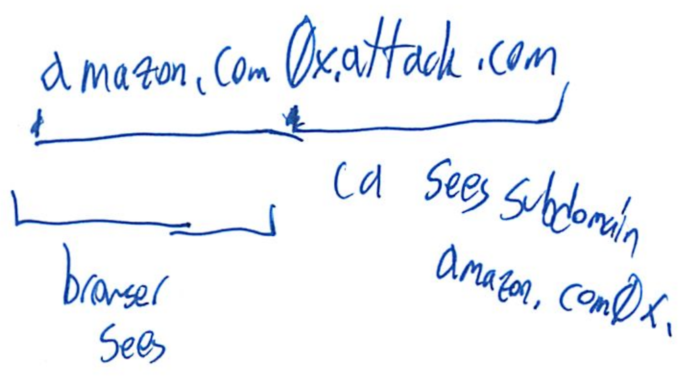
SSL certs



Some browsers used strcmp(s_1, s_2)

- C is null terminated (0-byte)
- but this was len bytes long
- One CA used Java - encoded 0 bytes inside

- So got amazon.com∅x.attack.com



⑩

Missing ACL checks

paymaxx.com / print.cgi ? id = 7235...
(user id)

So can just change user id in URL

No checking log in page

Debian's OpenSSL

```
Rand...()  {
    int x; //not initalized
    :
    :
    return x^ ... ^ ---;
}
```

assumed noise on stack

Some script cleaned up this ~~code~~ unused variable

took a while to notice
- several years
- everything worked

(11)

Principle : Economy of mechanism

- reuse existing, tested-by-time mechanism

- OS security
- Network firewalls
- Crypto

- (can have extra defenses as long as you understand why/ what trying to solve

Secrecy

Should design/code secrecy be in threat model?

- Want to be able to share to validate
- One more assumption you are relying on
    - trust more intermediate things
    - more/harder to enforce

(12)

- harder to reserve in future / scalable
  - easier to change key than design

Should understand **why** system is secure
- 1 thing I can't reveal
- instead of protecting everything

Terminology

A is trusted - bad thing
  if A is not secure, you're screwed
  transitive trust - must trust everything downstream

A is trustworthy - good thing
  works ~~all~~ whether you need it or not

"Trusted Computing Base" [TCB]
- try to shrink
- OS, Boot loader, VMM

(13)

HW

Lab 1 posted today

Review Session Tmo 7PM

Read paper

9/5

```
Introduction
============

Administrivia
  Lectures will be MW11-12:30, in 4-237 (unless the registrar moves us again).
    Each lecture will cover a paper in systems security (except today).
      Preliminary paper list posted online, likely to change a bit.
      If you are interested in specific topics or papers, send us email.
    Read the paper before lecture.
    Turn in answers to a short homework question before lecture.
    Send email with a question about the paper; will try to answer in lecture.
    Will discuss the paper in class.
    Interrupt, ask questions, point out mistakes.
  Two quizzes during regular lecture time slot.
    No "final exam" during finals week; second quiz near end-of-term.
  Assignments: 6 labs + final project.
    Lab 1 out today: buffer overflows.  Start early.
    Labs will look like real-world systems, in some respects:
      Many interacting parts written in different languages.
      Will look at/write x86 asm, C, Python, Javascript, ..
    Final project at the end of the course (groups of 2-3 people).
    Presentations during the last week of class.
    Think of projects you'd like to work on as you're reading papers.
    OK to combine with other class projects or your own research.
  Tutorial on how to get started with the VM, start writing your exploit.
    Thursday (tomorrow) 7pm, room TBD.
  Two TAs: David, Taesoo.
  Sign up for Piazza (link on course web site).
    Use it to ask questions about labs, see what others are stuck on, etc.
    We will post any important announcements there.
  Warning about security work/research on MITnet (and in general).
    Know the rules: http://ist.mit.edu/services/athena/olh/rules.
    Just because something is technically possible, doesn't mean it's legal.
    Ask course staff for advice if in doubt.

What is security?
  Achieving some goal in the presence of an adversary.
    Many systems are connected to the internet, which has adversaries.
    Thus, design of many systems might need to address security.
      i.e., will the system work when there's an adversary?
  High-level plan for thinking about security:
    Policy: the goal you want to achieve.
      e.g. only Alice should read file F.
      Common goals: confidentiality, integrity, availability.
    Threat model: assumptions about what the attacker could do.
      e.g. can guess passwords, cannot physically grab file server.
      Better to err on the side of assuming attacker can do something.
    Mechanism: knobs that your system provides to help uphold policy.
      e.g. user accounts, passwords, file permissions, encryption.
  Resulting goal: no way for adversary within threat model to violate policy.
    Note that goal has nothing to say about mechanism.

Why is security hard?  Negative goal.
  Need to guarantee policy, assuming the threat model.
  Difficult to think of all possible ways that attacker might break in.
  Realistic threat models are open-ended (almost negative models).
  Contrast: easy to check whether a positive goal is upheld,
            e.g., Alice can actually read file F.
  Weakest link matters.
  Iterative process: design, update threat model as necessary, etc.

What's the point if we can't achieve perfect security?
```

```
  In reality, must often manage security risk vs benefit.
    More secure systems means less risk (or consequence) of some compromises.
    Insecure system may require manual auditing to check for attacks, etc.
  Better security often makes new functionality practical and safe.
    Suppose you want to run some application on your system.
    Large companies often prohibit users from installing software that
          hasn't been approved on their desktops, partly due to security.
    Javascript in the browser is isolated, making it ok (for the most part)
          to run new code/applications without manual inspection/approval.
    (or virtual machines, or Native Client, or better OS isolation mechanisms)
    Similarly, VPNs make it practical to mitigate risk of allowing employees
          to connect to a corporate network from anywhere on the Internet.

What goes wrong #1: problems with the policy.
  Example: Fairfax County, VA school system.
    Each user has a principal corresponding to them, files, and password.
      (Just to be clear: technical term, not the job of school principal)
    Student can access only his/her own files.
    Teacher can access only files of students in his/her class.
    Superintendent has access to everyone's files.
    Teachers can add students (principals) to their class.
    Teachers can change password of students in their class.
    What's the worst that could happen if student gets teacher's password?
    Policy amounts to: teachers can do anything.
  Example: Sarah Palin's email account.
    Yahoo email accounts have a username, password, and security questions.
    User can log in by supplying username and password.
    If user forgets password, can reset by answering security Qs.
    Security questions can sometimes be easier to guess than password.
    Some adversary guessed Sarah Palin's high school, birthday, etc.
    Policy amounts to: can log in with either password or security Qs.
      (no way to enforce "Only if user forgets password, then ...")
  Example: Amazon/Apple break-in.
    Amazon allows adding credit card, then use last 4 digits to reset password, ...
    http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking/all/
  How to solve?
    Think hard about implications of policy statements.
    Some policy checking tools can automate this process.
    Automation requires higher-level goal (e.g. no way for student to do X).

What goes wrong #2: problems with threat model / assumptions.
  Example: human factors not accounted for.
    Phishing attacks.
    User gets email asking to renew email account, transfer money, or ...
    Tech support gets call from convincing-sounding user to reset password.
    "Rubberhose cryptanalysis".
  Example: all SSL certificate CAs are fully trusted.
    To connect to an SSL-enabled web site, web browser verifies certificate.
    Certificate is a combination of server's host name and cryptographic key,
      signed by some trusted certificate authority (CA).
    Long list (hundreds) of certificate authorities trusted by most browsers.
    If any CA is compromised, adversary can intercept SSL connections with
      a "fake" certificate for any server host name.
    Last year, a Dutch CA was compromised, issued fake certs for
      many domains (google, yahoo, tor, ...), apparently used in Iran (?).
  Example: assuming good randomness for cryptography.
    Need high-quality randomness to generate the keys that can't be guessed.
    Problem: embedded devices, virtual machines may not have much randomness.
    As a result, many keys are similar or susceptible to guessing attacks.
    [ https://factorable.net/weakkeys12.extended.pdf ]
  Example: subverting military OS security.
    In the 80's, military encouraged research into secure OS'es.
    One unexpected way in which OS'es were compromised:
```

```
      adversary gained access to development systems, modified OS code.
  Example: subverting firewalls.
    Adversaries can connect to an unsecured wireless behind firewall
    Adversaries can trick user behind firewall to disable firewall
      Might suffice just to click on link http://firewall/?action=disable
      Or maybe buy an ad on CNN.com pointing to that URL (effectively)?
  Example: machines disconnected from the Internet are secure?
    Stuxnet worm spread via specially-constructed files on USB drives.
  How to solve?
    Think hard (unfortunately).
    Simpler, more general threat models.
    Better designs may eliminate / lessen reliance on certain assumptions.
      E.g., alternative trust models that don't have fully-trusted CAs.
      E.g., authentication mechanisms that aren't susceptible to phishing.

What goes wrong #3: problems with the mechanism -- bugs.
  Example: programming mistakes: buffer overflows, etc.
    Security-critical program manipulates strings in an unsafe way.

      int read_num(void) {
        char buf[128];
        gets(buf);
        return atoi(buf);
      }

    Adversary can manipulate inputs to run arbitrary code in this program.

  Example: integer overflows matter, in C code (e.g., malloc).
  Example: Moxie's SSL certificate name checking bug
    Null byte vs. length-encoding.
  Example: PayMaxx W2 form disclosure.
    Web site designed to allow users to download their tax forms online.
    Login page asks for username and password.
    If username and password OK, redirected to new page.
    Link to print W2 form was of the form:
      http://paymaxx.com/print.cgi?id=91281
    Turns out 91281 was the user's ID; print.cgi did not require password
    Can fetch any user's W2 form by going directly to the print.cgi URL
    Possibly a wrong threat model: doesn't match the real world?
      System is secure if adversary browses the web site through browser
      System not secure if adversary synthesizes new URLs on their own
    Hard to say if developers had wrong threat model, or buggy mechanism..
  Example: Debian PRNG weakness.
    Debian shipped with a library called OpenSSL for cryptography.
    Used to generate secret keys (for signing or encrypting things later).
    Secret key generated by gathering some random numbers.
    Developer accidentally "optimized away" part of random number generator.
    No-one noticed for a while, because could still generate secret keys.
    Problem: many secret keys were identical, and not so secret as a result.
  Example: bugs in sandbox (NaCl, Javascript).
    Allows adversary to escape isolation, do operations they weren't supposed to.

How to avoid mechanism problems?
  Use common, well-tested security mechanisms ("Economy of mechanism")
  Audit these common security mechanisms (lots of incentive to do so)
  Avoid developing new, one-off mechanisms that may have bugs
  Good mechanism supports many uses, policies (more incentive to audit)
  Examples of common mechanisms:
    - OS-level access control (but, could often be better)
    - network firewalls (but, could often be better)
    - cryptography, cryptographic protocols.

Open vs. closed design or mechanism
```

```
  Why not make everything closed or secret (design, impl, code, ...)?
  Threat model: best to have the most conservative threat model possible
    What if you assume your implementation or design are secret?
    If implementation is revealed, hard to change to re-gain security!
    Must re-implement or re-design system
    If only assumption was that password/key/... is secret, can change.
  What if you don't make assumptions about design/impl being secret?
    Often a good idea to publish design/impl to get more review.
    If others use your mechanism, it will be much better tested!
    Helps ensure you don't accidentally assume design/impl are secret.

Rely on less mechanism, by changing the threat model when possible.
  Terminology:
    (X) is trustworthy: means that (X) is worth trusting -- good!
      (X) is not buggy, will not be compromised, will not fail.
    (X) is trusted: means that (X) _must_ be trustworthy -- bad!
      Something will fail if (X) is buggy, compromised, or fails.
      Suppose you're editing a text file on some machine.
      The OS kernel is trusted by all parties (users, apps, etc).
      The text editor is trusted by you (but maybe not other users).
    "Trusted" is often transitive -- bad again!
      If you have to trust the text editor, and the text editor has
        to trust developers of some library it uses, then you may
        have no choice but to trust that library too.
  High-level plan:
    Reduce amount of trusted code.
      .. by enforcing policy using a more trustworthy mechanism.
      .. e.g. the OS kernel should let this app write to only one file?
      "Principle of least privilege".
      Doing this requires well-designed, flexible security mechanisms.
    Improve trustworthiness of trusted code.
      .. by auditing, program analysis, better languages, etc.

References
```



```
  http://catless.ncl.ac.uk/Risks/26.02.html#subj7.1
  http://en.wikipedia.org/wiki/Sarah_Palin_email_hack
  http://www.thinkcomputer.com/corporate/whitepapers/identitycrisis.pdf
  http://en.wikipedia.org/wiki/DigiNotar#Issuance_of_fraudulent_certificates
```

6.858 Tutorial 9/7

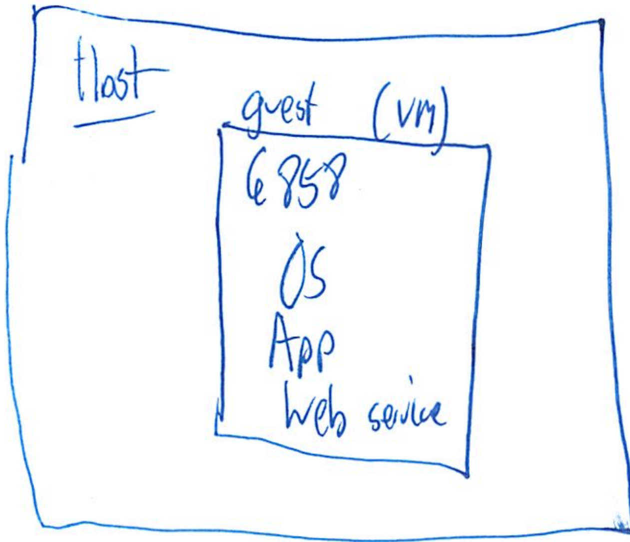
Download VM image and VMware

Windows: vmplayer, putty

Tools
- vmware
- qemu
- ssh, scp, sshfs
- git
- vi

Overview
- VM
- Zookws
- exploit → crash
- gdb
- submission

②



? run VM indentity on guest OS

- VM image
- Config file
  - L mac address
    - ram
    - CPU
- VMM
  - VMware
  - KVM
  - Virtual Box
- Don't do VM Tools!

3

Download lab

by $ git clone

and lab Folder

precompiled binary

bugs.txt is answer sheet

Clean env - consistent layout of tex

(i think he say)

Must make and launch

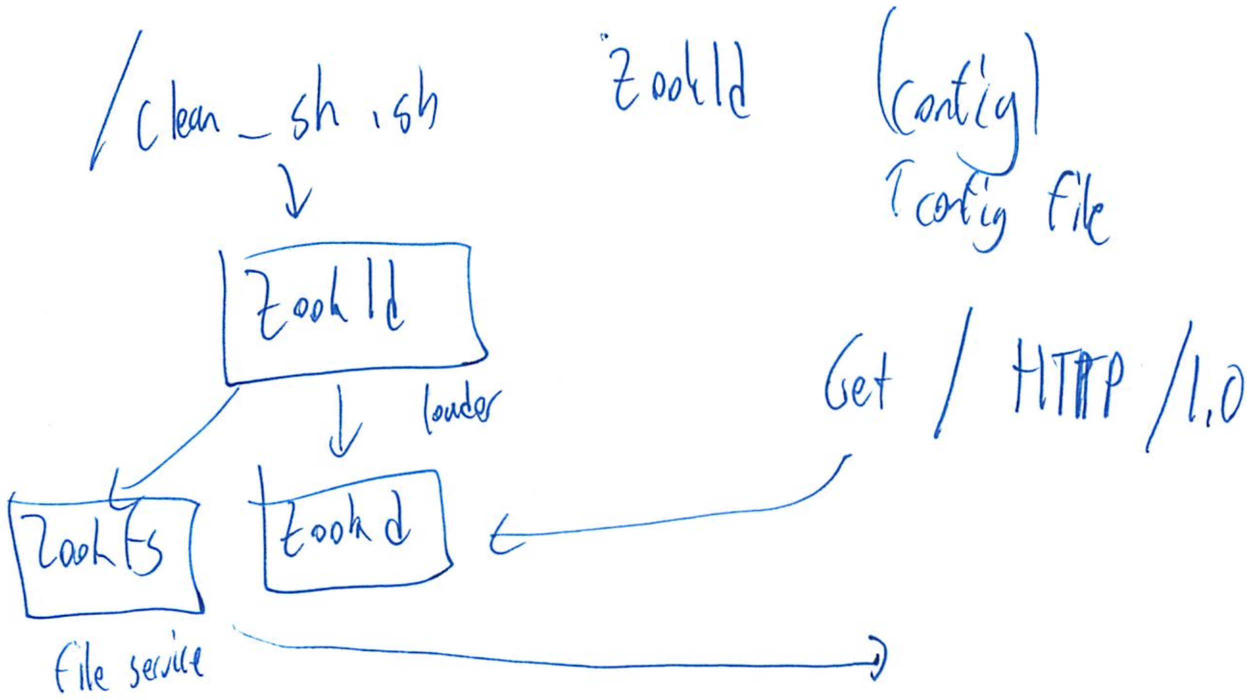
./clean-env.sh ./zookld zook-exstack.conf

ifconfig to find ip address

Connect from outside VM via web browser

zoobar

(4)



(TA not very cler)

Config

[name service]
port = 8080

[zookd]
Cmd = zookd-exstack
? w/ executable stack

Also a -ssp
? gcc compiles w/

5

- SSP has stack canary
  - └ described Mon

.c file for daemon
- fs
- d

httpd.c .h ← how to parse HTTP req
- CGI env

So in exploit-crash.py can send special request to server

ideally such that it crashes

Hard to work in VMware screen
SSH into instead

(6)

GDB lets you know where some buffers are

Details on Mon

Can't use print statements

So use debugger

attach GDB w/ a specified breakpoint

basic crash is enough for first point

then get info out of it

-g adds extra info to binary

-m32 is the type of binary

break [illegible] adds a breakpoint

list (something)

bt = back trace

└ prints layout of stack

up and down

(2)

info r gives you stack registers

print call tells you what it called

step = step forward next line (enter functions)

disassemble call = prints out the assembly code

si = stepi = one 1 instruction

run

continue

examine (x) = print memory

x/10x $esp = print 10 words from %esp (stack pointer)

$eip - try to disassemble from every step (??)

GDB is very configurable

---

gdb -p to attach to running process (process id)
 ↳ can only look at one at once

(8)

gdpush -p $(pgrep zookd-exstack)

Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors

*Periklis Akritidis,** Manuel Costa,*† Miguel Castro,*† Steven Hand**

**Computer Laboratory*
University of Cambridge, UK
{pa280,smh22}@cl.cam.ac.uk

†*Microsoft Research*
Cambridge, UK
{manuelc,mcastro}@microsoft.com

Abstract

Attacks that exploit out-of-bounds errors in C and C++ programs are still prevalent despite many years of research on bounds checking. Previous backwards compatible bounds checking techniques, which can be applied to unmodified C and C++ programs, maintain a data structure with the bounds for each allocated object and perform lookups in this data structure to check if pointers remain within bounds. This data structure can grow large and the lookups are expensive.

In this paper we present a backwards compatible bounds checking technique that substantially reduces performance overhead. The key insight is to constrain the sizes of allocated memory regions and their alignment to enable efficient bounds lookups and hence efficient bounds checks at runtime. Our technique has low overhead in practice—only 8% throughput decrease for Apache—and is more than two times faster than the fastest previous technique and about five times faster—using less memory—than recording object bounds using a splay tree.

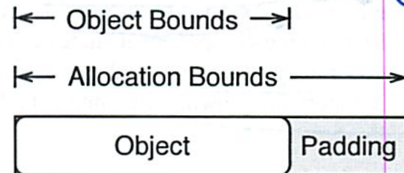


Figure 1: Allocated memory is often padded to a particular alignment boundary, and hence can be larger than the requested object size. By checking *allocation bounds* rather than object bounds, we allow benign accesses to the padding, but can significantly reduce the cost of bounds lookups at runtime.

1 Introduction

Bounds checking C and C++ code protects against a wide range of common vulnerabilities. The challenge has been making bounds checking fast enough for production use and at the same time backwards compatible with binary libraries to allow incremental deployment. Solutions using *fat pointers* [24, 18] extend the pointer representation with bounds information. This enables efficient bounds checks but breaks backwards compatibility because increasing the pointer size changes the memory layout of data structures. Backwards compatible bounds checking techniques [19, 30, 36, 15] use a separate data structure to lookup bounds information. Initial attempts incurred a significant overhead [19, 30, 36] (typically 2x–10x) because looking up bounds is expensive and the data structure can grow large. More recent work [15] has applied sophisticated static pointer analysis to reduce the number of bounds lookups; this managed to reduce the runtime overhead on the Olden benchmarks to 12% on average.

In this paper we present *baggy bounds checking*, a backwards compatible bounds checking technique that reduces the cost of bounds checks. We achieve this by enforcing *allocation bounds* rather than precise object bounds, as shown in Figure 1. Since memory allocators pad object allocations to align the pointers they return, there is a class of benign out-of-bounds errors that violate the object bounds but fall within the allocation bounds. Previous work [4, 19, 2] has exploited this property in a variety of ways.

Here we apply it to efficient backwards compatible bounds checking. We use a binary buddy allocator to enable a compact representation of the allocation bounds: since all allocation sizes are powers of two, a single byte is sufficient to store the binary logarithm of the allocation

size. Furthermore, there is no need to store additional information because the base address of an allocation with size s can be computed by clearing the $log_2(s)$ least significant bits of any pointer to the allocated region. This allows us to use a space and time efficient data structure for the bounds table. We use a contiguous array instead of a more expensive data structure (such as the splay trees used in previous work). It also provides us with an elegant way to deal with common cases of temporarily out-of-bounds pointers. We describe our design in more detail in Section 2.

We implemented *baggy bounds checking* as a compiler plug-in for the Microsoft Phoenix [22] code generation framework, along with additional run time components (Section 3). The plug-in inserts code to check bounds for all pointer arithmetic that cannot be statically proven safe, and to align and pad stack variables where necessary. The run time component includes a binary buddy allocator for heap allocations, and user-space virtual memory handlers for growing the bounds table on demand.

In Section 4 we evaluate the performance of our system using the Olden benchmark (to enable a direct comparison with Dhurjati and Adve [15]) and SPECINT 2000. We compare our space overhead with a version of our system that uses the splay tree implementation from [19, 30]. We also verify the efficacy of our system in preventing attacks using the test suite described in [34], and run a number of security critical COTS components to confirm its applicability.

Section 5 describes our design and implementation for 64-bit architectures. These architectures typically have "spare" bits within pointers, and we describe a scheme that uses these to encode bounds information directly in the pointer rather than using a separate lookup table. Our comparative evaluation shows that the performance benefit of using these spare bits to encode bounds may not in general justify the additional complexity; however using them just to encode information to recover the bounds for out-of-bounds pointers may be worthwhile.

Finally we survey related work (Section 6), discuss limitations and possible future work (Section 7) and conclude (Section 8).

2 Design

2.1 Baggy Bounds Checking

Our system shares the overall architecture of backwards compatible bounds checking systems for C/C++ (Figure 2).

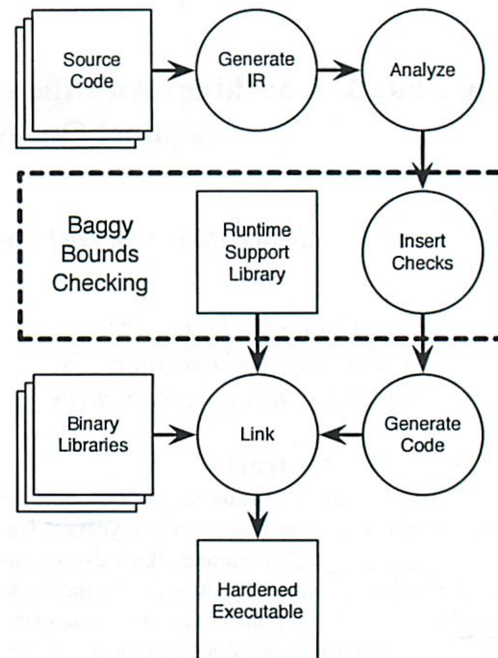


Figure 2: Overall system architecture, with our contribution highlighted within the dashed box.

It converts source code to an intermediate representation (IR), finds potentially unsafe pointer arithmetic operations, and inserts checks to ensure their results are within bounds. Then, it links the generated code with our runtime library and binary libraries—compiled with or without checks—to create a hardened executable.

We use the *referent object* approach for bounds checking introduced by Jones and Kelly [19]. Given an in-bounds pointer to an object, this approach ensures that any derived pointer points to the same object. It records bounds information for each object in a *bounds table*. This table is updated on allocation and deallocation of objects: this is done by the `malloc` family of functions for heap-based objects; on function entry and exit for stack-based objects; and on program startup for global objects.

The referent object approach performs bounds checks on pointer arithmetic. It uses the source pointer to lookup the bounds in the table, performs the operation, and checks if the destination pointer remains in bounds. If the destination pointer does not point to the same object, we mark it out-of-bounds to prevent any dereference (as in [30, 15]). However we permit its use in further pointer arithmetic, since it may ultimately result in an in-bounds pointer. The marking mechanism is described in detail in Section 2.4.

Baggy bounds checking uses a very compact repre-

sentation for bounds information. Previous techniques recorded a pointer to the start of the object and its size in the bounds table, which requires at least eight bytes. We pad and align objects to powers of two and enforce allocation bounds instead of object bounds. This enables us to use a single byte to encode bounds information. We store the binary logarithm of the allocation size in the bounds table:

```
e = log2(size);
```

Given this information, we can recover the allocation size and a pointer to the start of the allocation with:

```
size = 1 << e;

base = p & ~(size-1);
```

To convert from an in-bounds pointer to the bounds for the object we require a *bounds table*. Previous solutions based on the referent object approach (such as [19, 30, 15]) have implemented the bounds table using a splay tree.

Baggy bounds, by contrast, implement the bounds table using a contiguous array. The table is small because each entry uses a single byte. Additionally, we partition memory into aligned *slots* with *slot_size* bytes. The bounds table has an entry for each slot rather than an entry per byte. So the space overhead of the table is $1/slot_size$, and we can tune *slot_size* to balance memory waste between padding and table size. We align objects to slot boundaries to ensure that no two objects share a slot.

Accesses to the table are fast. To obtain a pointer to the entry corresponding to an address, we right-shift the address by the constant $log_2(slot_size)$ and add the constant table base. We can use this pointer to retrieve the bounds information with a single memory access, instead of having to traverse and splay a splay tree (as in previous solutions).

Note that baggy bounds checking permits benign out-of-bounds accesses to the memory padding after an object. This does not compromise security because these accesses cannot write or read other objects. They cannot be exploited for typical attacks such as (*a*) overwriting a return address, function pointer or other security critical data; or (*b*) reading sensitive information from another object, such as a password.

We also defend against a less obvious attack where the program reads values from the padding area that were originally written to a deleted object that occupied the same memory. We prevent this attack by clearing the padding on memory allocation.

Pointer arithmetic operation:

```
p' = p + i
```

Explicit bounds check:

```
size = 1 << table[p>>slot_size]
base = p & ~(size-1)

p' >= base && p' - base < size
```

Optimized bounds check:

```
(p^p')>>table[p>>slot_size] == 0
```

Figure 3: Baggy bounds enables optimized bounds checks: we can verify that pointer p' derived from pointer p is within bounds by simply checking that p and p' have the same prefix with only the e least significant bits modified, where e is the binary logarithm of the allocation size.

2.2 Efficient Checks

In general, bounds checking the result p' of pointer arithmetic on p involves two comparisons: one against the lower bound and one against the upper bound, as shown in Figure 3.

We devised an optimized bounds check that does not even need to compute the lower and upper bounds. It uses the value of p and the value of the binary logarithm of the allocation size, e, retrieved from the bounds table. The constraints on allocation size and alignment ensure that p' is within the allocation bounds if it differs from p only in the e least significant bits. Therefore, it is sufficient to shift p^p' by e and check if the result is zero, as shown in Figure 3.

Furthermore, for pointers p' where `sizeof(*p') > 1`, we also need to check that `(char *) p' + sizeof(*p') - 1` is within bounds to prevent a subsequent access to `*p'` from crossing the allocation bounds. Baggy bounds checking can avoid this extra check if p' points to a built-in type. Aligned accesses to these types cannot overlap an allocation boundary because their size is a power of two and is less than *slot_size*. When checking pointers to structures that do not satisfy these constraints, we perform both checks.

2.3 Interoperability

Baggy bounds checking works even when instrumented code is linked against libraries that are not instrumented.

The library code works without change because it performs no checks but it is necessary to ensure that instrumented code works when accessing memory allocated in an uninstrumented library. This form of interoperability is important because some libraries are distributed in binary form.

We achieve interoperability by using the binary logarithm of the maximum allocation size as the default value for bounds table entries. Instrumented code overwrites the default value on allocations with the logarithm of the allocation size and restores the default value on deallocations. This ensures that table entries for objects allocated in uninstrumented libraries inherit the default value. Therefore, instrumented code can perform checks as normal when accessing memory allocated in a library, but checking is effectively disabled for these accesses. We could intercept heap allocations in library code at link time and use the buddy allocator to enable bounds checks on accesses to library-allocated memory, but this is not done in the current prototype.

2.4 Support for Out-Of-Bounds Pointers

A pointer may legally point outside the object bounds in C. Such pointers should not be dereferenced but can be compared and used in pointer arithmetic that can eventually result in a valid pointer that may be dereferenced by the program.

Out-of-bounds pointers present a challenge for the referent object approach because it relies on an in-bounds pointer to retrieve the object bounds. The C standard only allows out-of-bounds pointers to one element past the end of an array. Jones and Kelly [19] support these legal out-of-bounds pointers by padding objects with one byte. We did not use this technique because it interacts poorly with our constraints on allocation sizes: adding one byte to an allocation can double the allocated size in the common case where the requested allocation size is a power of two.

Many programs violate the C standard and generate illegal but harmless out-of-bounds pointers that they never dereference. Examples include faking a base one array by decrementing the pointer returned by `malloc` and other equally tasteless uses. CRED [30] improved on the Jones and Kelly bounds checker [19] by tracking such pointers using another auxiliary data structure. We did not use this approach because it adds overhead on deallocations of heap and local objects: when an object is deallocated the auxiliary data structure must be searched to remove entries tracking out-of-bounds pointers to the object. Additionally, entries in this auxiliary data structure may accumulate until their referent object is deallocated.

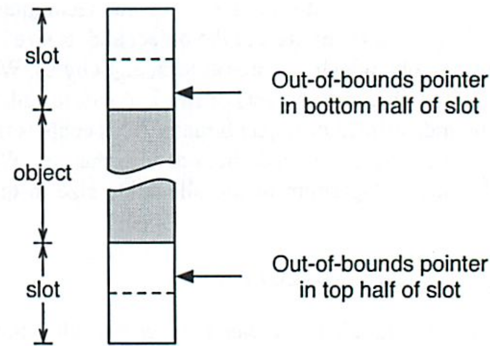


Figure 4: We can tell whether a pointer that is out-of-bounds by less than $slot_size/2$ is below or above an allocation. This lets us correctly adjust it to get a pointer to the object by respectively adding or subtracting $slot_size$.

We handle out-of-bounds pointers within $slot_size/2$ bytes from the original object as follows. First, we mark out-of-bounds pointers to prevent them from being dereferenced (as in [15]). We use the memory protection hardware to prevent dereferences by setting the most significant bit in these pointers and by restricting the program to the lower half of the address space (this is often already the case for user-space programs). We can recover the original pointer by clearing the bit.

The next challenge is to recover a pointer to the referent object from the out-of-bounds pointer without resorting to an additional data structure. We can do this for the common case when out-of-bounds pointers are at most $slot_size/2$ bytes before or after the allocation. Since the allocation bounds are aligned to slot boundaries, we can find if a marked pointer is below or above the allocation by checking whether it lies in the top or bottom half of a memory slot respectively, as illustrated in Figure 4. We can recover a pointer to the referent object by adding or subtracting $slot_size$ bytes. This technique cannot handle pointers that go more than $slot_size/2$ bytes outside the original object. In Section 5.2, we show how to take advantage of the spare bits in pointers on 64 bit architectures to increase this range, and in Section 7 we discuss how we could add support for arbitrary out-of-bounds pointers while avoiding some of the problems of previous solutions.

It is not necessary to instrument pointer dereferences. Similarly, there is no need to instrument pointer equality comparisons because the comparison will be correct whether the pointers are out-of-bounds or not. But we need to instrument inequality comparisons to support

comparing an out-of-bounds pointer with an in-bounds one: the instrumentation must clear the high-order bit of the pointers before comparing them. We also instrument pointer differences in the same way.

Like previous bounds checking solutions [19, 30, 15], we do not support passing an out-of-bounds pointer to uninstrumented code. However, this case is rare. Previous work [30] did not encounter this case in several million lines of code.

2.5 Static Analysis

Bounds checking has relied heavily on static analysis to optimize performance [15]. Checks can be eliminated if it can be statically determined that a pointer is safe, i.e. always within bounds, or that a check is redundant due to a previous check. Furthermore, checks or just the bounds lookup can be hoisted out of loops. We have not implemented a sophisticated analysis and, instead, focused on making checks efficient.

Nevertheless, our prototype implements a simple intra-procedural analysis to detect safe pointer operations. We track allocation sizes and use the compiler's variable range analysis to eliminate checks that are statically shown to be within bounds. We also investigate an approach to hoist checks out of loops that is described in Section 3.

We also use static analysis to reduce the number of local variables that are padded and aligned. We only pad and align local variables that are indexed unsafely within the function, or whose address is taken, and therefore possibly leaked from the function. We call these variables *unsafe*.

3 Implementation

We used the Microsoft Phoenix [22] code generation framework to implement a prototype system for x86 machines running Microsoft Windows. The system consists of a plug-in to the Phoenix compiler and a runtime support library. In the rest of this section, we describe some implementation details.

3.1 Bounds Table

We chose a $slot_size$ of 16 bytes to avoid penalizing small allocations. Therefore, we reserve $1/16^{th}$ of the address space for the bounds table. Since pages are allocated to the table on demand, this increases memory utilization by only 6.25%. We reserve the address space required for the bounds table on program startup and install a user space page fault handler to allocate missing table pages on demand. All the bytes in these pages are initialized by the handler to the value 31, which encompasses all the addressable memory in the x86 (an allocation size of 2^{31} at base address 0). This prevents out-of-bounds errors when instrumented code accesses memory allocated by uninstrumented code.

3.2 Padding and Aligning

We use a binary buddy allocator to satisfy the size and alignment constraints on heap allocations. Binary buddy allocators provide low external fragmentation but suffer from internal fragmentation because they round allocation sizes to powers of two. This shortcoming is put to good use in our system. Our buddy allocator implementation supports a minimum allocation size of 16 bytes, which matches our $slot_size$ parameter, to ensure that no two objects share the same slot.

We instrument the program to use our version of `malloc`-style heap allocation functions based on the buddy allocator. These functions set the corresponding bounds table entries and zero the padding area after an object. For local variables, we align the stack frames of functions that contain unsafe local variables at runtime and we instrument the function entry to zero the padding and update the appropriate bounds table entries. We also instrument function exit to reset table entries to 31 for interoperability when uninstrumented code reuses stack memory. We align and pad static variables at compile time and their bounds table entries are initialized when the program starts up.

Unsafe function arguments are problematic because padding and aligning them would violate the calling convention. Instead, we copy them on function entry to appropriately aligned and padded local variables and we change all references to use the copies (except for uses of `va_list` that need the address of the last explicit argument to correctly extract subsequent arguments). This preserves the calling convention while enabling bounds checking for function arguments.

The Windows runtime cannot align stack objects to more than 8K nor static objects to more than 4K (configurable using the `/ALIGN` linker switch). We could replace these large stack and static allocations with heap allocations to remove this limitation but our current prototype sets the bounds table entries for these objects to 31.

Zeroing the padding after an object can increase space and time overhead for large padding areas. We avoid this

overhead by relying on the operating system to zero allocated pages on demand. Then we track the subset of these pages that is modified and we zero padding areas in these pages on allocations. Similar issues are discussed in [9] and the standard allocator uses a similar technique for `calloc`. Our buddy allocator also uses this technique to avoid explicitly zeroing large memory areas allocated with `calloc`.

3.3 Checks

We add checks for each pointer arithmetic and array indexing operation but, following [15], we do not instrument accesses to scalar fields in structures and we do not check pointer dereferences. This facilitates a direct comparison with [15]. We could easily modify our implementation to perform these checks, for example, using the technique described in [14].

We optimize bounds checks for the common case of in-bounds pointers. To avoid checking if a pointer is marked out-of-bounds in the fast path, we set all the entries in the bounds table that correspond to out-of-bounds pointers to zero. Since out-of-bounds pointers have their most significant bit set, we implement this by mapping all the virtual memory pages in the top half of the bounds table to a shared zero page. This ensures that our slow path handler is invoked on any arithmetic operation involving a pointer marked out-of-bounds.

```
bounds lookup       mov eax, buf
                    shr eax, 4
                    mov al, byte ptr [TABLE+eax]

pointer arithmetic      char *p = buf[i];

bounds check        mov ebx, buf
                    xor ebx, p
                    shr ebx, al
                    jz ok
                      p = slowPath(buf, p)
                    ok:
```

Figure 5: Code sequence inserted to check unsafe pointer arithmetic.

Figure 5 shows the `x86` code sequence that we insert before an example pointer arithmetic operation. First, the source pointer, `buf`, is right shifted to obtain the index of the bounds table entry for the corresponding slot. Then the logarithm of the allocation size e is loaded from the bounds table into register `al`. The result of the pointer arithmetic, `p`, is xored with the source pointer, `buf`, and right shifted by `al` to discard the bottom bits. If `buf` and `p` are both within the allocation bounds they can only differ in the $log_2 e$ least significant bits (as discussed before). So if the zero flag is set, `p` is within the allocation bounds. Otherwise, the `slowPath` function is called.

The `slowPath` function starts by checking if `buf` has been marked out-of-bounds. In this case, it obtains the referent object as described in 2.4, resets the most significant bit in `p`, and returns the result if it is within bounds. Otherwise, the result is out-of-bounds. If the result is out-of-bounds by more than half a slot, the function signals an error. Otherwise, it marks the result out-of-bounds and returns it. Any attempt to dereference the returned pointer will trigger an exception. To avoid disturbing register allocation in the fast path, the `slowPath` function uses a special calling convention that saves and restores all registers.

As discussed in Section 3.3, we must add `sizeof(*p)` to the result and perform a second check if the pointer is not a pointer to a built-in type. In this case, `buf` is a `char*`.

Similar to previous work, we provide bounds checking wrappers for Standard C Library functions such as `strcpy` and `memcpy` that operate on pointers. We replace during instrumentation calls to these functions with calls to their wrappers.

3.4 Optimizations

Typical optimizations used with bounds checking include eliminating redundant checks, hoisting checks out of loops, or hoisting just bounds table lookups out of loops. Optimization of inner loops can have a dramatic impact on performance. We experimented with hoisting bounds table lookups out of loops when all accesses inside a loop body are to the same object. Unfortunately, performance did not improve significantly, probably because our bounds lookups are inexpensive and hoisting can adversely effect register allocation.

Hoisting the whole check out of a loop is preferable when static analysis can determine symbolic bounds on the pointer values in the loop body. However, hoisting out the check is only possible if the analysis can determine that these bounds are guaranteed to be reached in every execution. Figure 6 shows an example where the loop bounds are easy to determine but the loop may terminate before reaching the upper bound. Hoisting out the check would trigger a false alarm in runs where the loop exits before violating the bounds.

We experimented with an approach that generates two versions of the loop code, one with checks and one without. We switch between the two versions on loop entry.

In the example of Figure 6, we lookup the bounds of p and if n does not exceed the size we run the unchecked version of the loop. Otherwise, we run the checked version.

```
for (i = 0; i < n; i++) {
    if (p[i] == 0) break;
    ASSERT(IN_BOUNDS(p, &p[i]));
    p[i] = 0;
}
```

↓

```
if (IN_BOUNDS(p, &p[n-1])) {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        p[i] = 0;
    }
} else {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        ASSERT(IN_BOUNDS(p, &p[i]));
        p[i] = 0;
    }
}
```

Figure 6: The compiler's range analysis can determine that the range of variable i is at most $0 \ldots n-1$. However, the loop may exit before i reaches $n-1$. To prevent erroneously raising an error, we fall back to an instrumented version of the loop if the hoisted check fails.

4 Experimental Evaluation

In this section we evaluate the performance of our system using CPU intensive benchmarks, its effectiveness in preventing attacks using a buffer overflow suite, and its usability by building and measuring the performance of real world security critical code.

4.1 Performance

We evaluate the time and peak memory overhead of our system using the Olden benchmarks and SPECINT 2000. We chose these benchmarks in part to allow a comparison against results reported for some other solutions [15, 36, 23]. In addition, to enable a more detailed comparison with splay-tree-based approaches—including measuring their space overhead—we implemented a variant of our approach which uses the splay tree code from previous systems [19, 30]. This implementation uses the standard allocator and is lacking support for illegal out-of-bounds pointers, but is otherwise identical to our system. We compiled all benchmarks with the Phoenix compiler using `/O2` optimization level and ran them on a 2.33 GHz Intel Core 2 Duo processor with 2 GB of RAM.

From SPECINT 2000 we excluded `eon` since it uses C++ which we do not yet support. For our splay-tree-based implementation only we did not run `vpr` due to its lack of support for illegal out-of-bounds pointers. We also could not run `gcc` because of code that subtracted a pointer from a NULL pointer and subtracted the result from NULL again to recover the pointer. Running this would require more comprehensive support for out-of-bounds pointers (such as that described in [30], as we propose in Section 7).

We made the following modifications to some of the benchmarks: First, we modified `parser` from SPECINT 2000 to fix an overflow that triggered a bound error when using the splay tree. It did not trigger an error with baggy bounds checking because in our runs the overflow was entirely contained in the allocation, but should it overlap another object during a run, the baggy checking would detect it. The unchecked program also survived our runs because the object was small enough for the overflow to be contained even in the padding added by the standard allocator.

Then, we had to modify `perlbmk` by changing two lines to prevent an out-of-bounds arithmetic whose result is never used and `gap` by changing 5 lines to avoid an out-of-bounds pointer. Both cases can be handled by the extension described in Section 5, but are not covered by the small out-of-bounds range supported by our 32-bit implementation and the splay-tree-based implementation.

Finally, we modified `mst` from Olden to disable a custom allocator that allocates 32 Kbyte chunks of memory at a time that are then broken down to 12 byte objects. This increases protection at the cost of memory allocation overhead and removes an unfair advantage for the splay tree whose time and space overheads are minimized when the tree contains just a few nodes, as well as baggy space overhead that benefits from the power of two allocation. This issue, shared with other systems offering protection at the memory block level [19, 30, 36, 15, 2], illustrates a frequent situation in C programs that may require tweaking memory allocation routines in the source code to take full advantage of checking. In this case merely changing a macro definition was sufficient.

We first ran the benchmarks replacing the standard allocator with our buddy system allocator to isolate its effects on performance, and then we ran them using our full system. For the Olden benchmarks, Figure 7 shows the execution time and Figure 8 the peak memory usage.

In Figure 7 we observe that some benchmarks in the Olden suite (`mst`, `health`) run significantly faster with

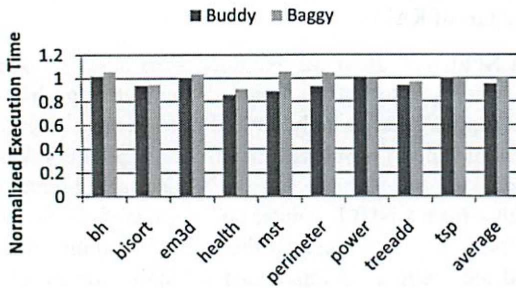


Figure 7: Execution time for the Olden benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

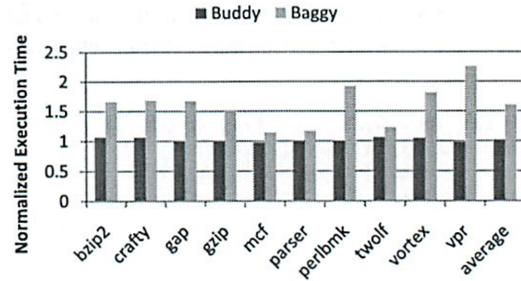


Figure 9: Execution time for SPECINT 2000 benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

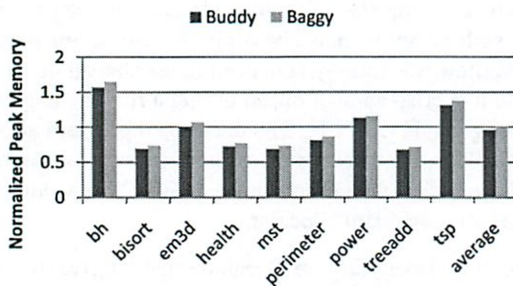


Figure 8: Peak memory use with the buddy allocator alone and with the full system for the Olden benchmarks, normalized by peak memory using the standard allocator without instrumentation.

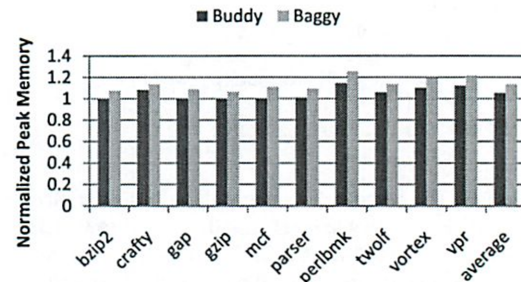


Figure 10: Peak memory use with the buddy allocator alone and with the full system for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

the buddy allocator than with the standard one. These benchmarks are memory intensive and any memory savings reflect on the running time. In Figure 8 we can see that the buddy system uses less memory for these than the standard allocator. This is because these benchmarks contain numerous small allocations for which the padding to satisfy alignment requirements and the per-allocation metadata used by the standard allocator exceed the internal fragmentation of the buddy system.

This means that the average time overhead of the full system across the entire Olden suite is actually zero, because the positive effects of using the buddy allocator mask the costs of checks. The time overhead of the checks alone as measured against the buddy allocator as a baseline is 6%. The overhead of the fastest previous bounds checking system [15] on the same benchmarks and same protection (modulo allocation vs. object bounds) is 12%, but their system also benefits from the technique of pool allocation which can also be used independently. Based on the breakdown of results reported in [15], their overhead measured against the pool allocation is 15%, and it seems more reasonable to compare these two numbers, as both the buddy allocator and pool allocation can be in principle applied independently on either system.

Next we measured the system using the SPECINT 2000 benchmarks. Figures 9 and 10 show the time and space overheads for SPECINT 2000 benchmarks.

We observe that the use of the buddy system has little effect on performance in average. The average runtime overhead of the full system with the benchmarks from SPECINT 2000 is 60%. `vpr` has the highest overhead of 127% because its frequent use of illegal pointers to fake base-one arrays invokes our slow path. We observed that adjusting the allocator to pad each allocation with 8 bytes from below, decreases the time overhead to 53% with only 5% added to the memory usage, although in general we are not interested in tuning the benchmarks like this. Interestingly, the overhead for `mcf` is a mere 16% compared to the 185% in [36] but the overhead of `gzip` is 55% compared to 15% in [36]. Such differences in performance are due to different levels of protection such as checking structure field indexing and checking dereferences, the effectiveness of different static analysis implementations in optimizing away checks, and the

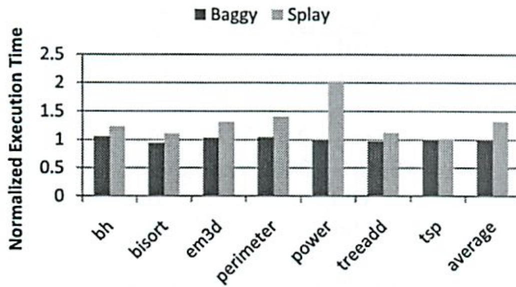


Figure 11: Execution time of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by the execution time using the standard system allocator without instrumentation. Benchmarks `mst` and `health` used too much memory and thrashed so their execution times are excluded.

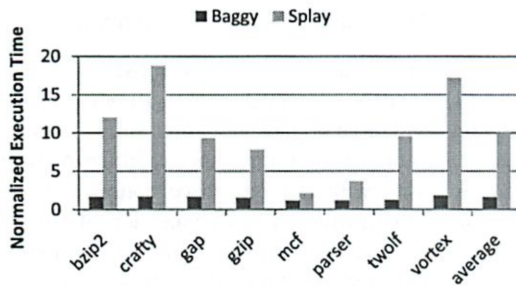


Figure 12: Execution time of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by the execution time using the standard system allocator without instrumentation.

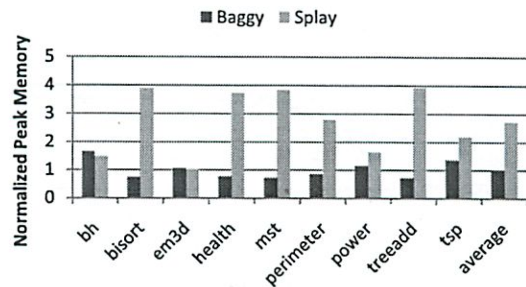


Figure 13: Peak memory use of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by peak memory using the standard allocator without instrumentation.

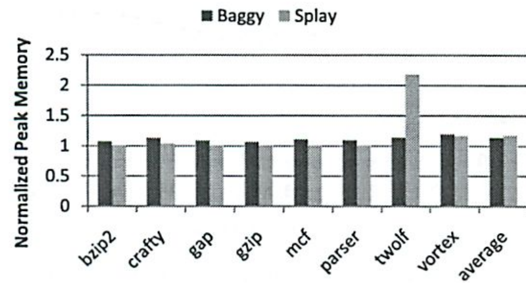


Figure 14: Peak memory use of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

different compilers used.

To isolate these effects, we also measured our system using the standard memory allocator and the splay tree implementation from previous systems [19, 30]. Figure 11 shows the time overhead for baggy bounds versus using a splay tree for the Olden benchmarks. The splay tree runs out of physical memory for the last two Olden benchmarks (`mst, health`) and slows down to a crawl, so we exclude them from the average of 30% for the splay tree. Figure 12 compares the time overhead against using a splay tree for the SPECINT 2000 benchmarks. The overhead of the splay tree exceeds 100% for all benchmarks, with an average of 900% compared to the average of 60% for baggy bounds checking.

Perhaps the most interesting result of our evaluation was space overhead. Previous solutions [19, 30, 15] do not report on the memory overheads of using splay trees, so we measured the memory overhead of our system using splay trees and compared it with the memory overhead of baggy bounds. Figure 13 shows that our system had negligible memory overhead for Olden, as opposed to the splay tree version's 170% overhead. Clearly Olden's numerous small allocations stress the splay tree by forcing it to allocate an entry for each.

Indeed, we see in Figure 14 that its space overhead for most SPECINT 2000 benchmarks is very low. Nevertheless, the overhead of 15% for baggy bounds is less than the 20% average of the splay tree. Furthermore, the potential worst case of double the memory was not encountered for baggy bounds in any of our experiments, while the splay tree did exhibit greater than 100% overhead for one benchmark (`twolf`).

The memory overhead is also low, as expected, compared to approaches that track meta data for each pointer. Xu *et al.* [36] report 331% for Olden, and Nagarakatte *et al.* [23] report an average of 87% using a hash-table (and 64% using a contiguous array) over Olden and a subset of SPECINT and SPECFP, but more than about 260% (or about 170% using the array) for the pointer intensive Olden benchmarks alone. These systems suffer memory overheads per pointer in order to provide optional temporal protection [36] and sub-object protection [23] and

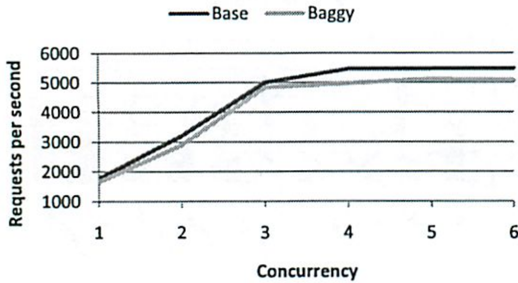


Figure 15: Throughput of Apache web server for varying numbers of concurrent requests.

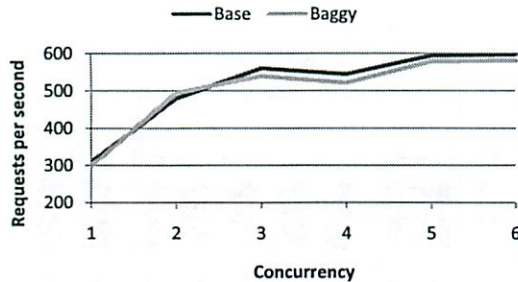


Figure 16: Throughput of NullHTTPD web server for varying numbers of concurrent requests.

it is interesting to contrast with them although they are not directly comparable.

4.2 Effectiveness

We evaluated the effectiveness of our system in preventing buffer overflows using the benchmark suite from [34]. The attacks required tuning to have any chance of success, because our system changes the stack frame layout and copies unsafe function arguments to local variables, but the benchmarks use the address of the first function argument to find the location of the return address they aim to overwrite.

Baggy bounds checking prevented 17 out of 18 buffer overflows in the suite. It failed, however, to prevent the overflow of an array inside a structure from overwriting a pointer inside the same structure. This limitation is also shared with other systems that detect memory errors at the level of memory blocks [19, 30, 36, 15].

4.3 Security Critical COTS Applications

Finally, to verify the usability of our approach, we built and measured a few additional larger and security critical

Program	KSLOC
openssl-0.9.8k	397
Apache-2.2.11	474
nullhttpd-0.5.1	2
libpng-1.2.5	36
SPECINT 2000	309
Olden	6
Total	1224

Table 1: Source lines of code in programs successfully built and run with baggy bounds.

COTS applications. Table 1 lists the total number of lines compiled in our experiments.

We built the OpenSSL toolkit version 0.9.8k [28] comprised of about 400 KSLOC, and executed its test suite measuring 10% time and 11% memory overhead.

Then we built and measured two web servers, Apache [31] and NullHTTPD [27]. Running NullHTTPD revealed three bounds violations similar to, and including, the one reported in [8]. We used the Apache benchmark utility with the keep-alive option to compare the throughput over a LAN connection of the instrumented and uninstrumented versions of both web servers. We managed to saturate the CPU by using the keep-alive option of the benchmarking utility to reuse connections for subsequent requests. We issued repeated requests for the servers' default pages and varied the number of concurrent clients until the throughput of the uninstrumented version leveled off (Figures 15 and 16). We verified that the server's CPU was saturated at this point, and measured a throughput decrease of 8% for Apache and 3% for NullHTTPD.

Finally, we built `libpng`, a notoriously vulnerability prone library that is widely used. We successfully ran its test program for 1000 PNG files between 1–2K found on a desktop machine, and measured an average runtime overhead of 4% and a peak memory overhead of 3.5%.

5 64-bit Architectures

In this section we verify and investigate ways to optimize our approach on 64 bit architectures. The key observation is that pointers in 64 bit architectures have spare bits to use. In Figure 17 (a) and (b) we see that current models of AMD64 processors use 48 out of 64 bits in pointers, and Windows further limit this to 43 bits for user space programs. Thus 21 bits in the pointer representation are not used. Next we describe two uses for these spare bits, and present a performance evaluation on AMD64.

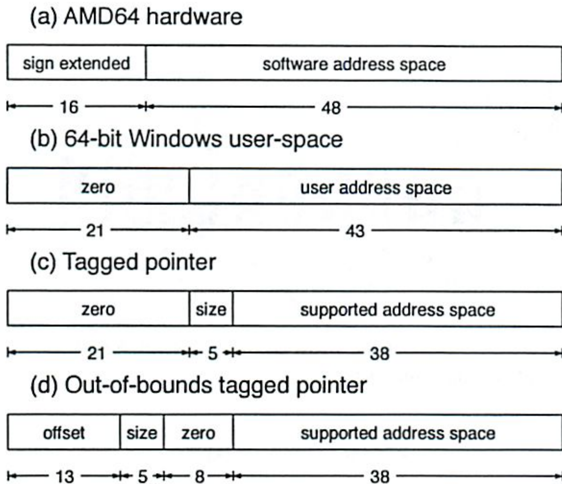


Figure 17: Use of pointer bits by AMD64 hardware, Windows applications, and baggy bounds tagged pointers.

5.1 Size Tagging

Since baggy bounds occupy less than a byte, they can fit in a 64 bit pointer's spare bits, removing the need for a separate data structure. These *tagged pointers* are similar to fat pointers in changing the pointer representation but have several advantages.

First, tagged pointers retain the size of regular pointers, avoiding fat pointers' register and memory waste. Moreover, their memory stores and loads are atomic, unlike fat pointers that break code relying on this. Finally, they preserve the memory layout of structures, overcoming the main drawback of fat pointers that breaks their interoperability with uninstrumented code.

For interoperability, we must also enable instrumented code to use pointers from uninstrumented code and vice versa. We achieve the former by interpreting the default zero value found in unused pointer bits as maximal bounds, so checks on pointers missing bounds succeed. The other direction is harder because we must avoid raising a hardware exception when uninstrumented code dereferences a tagged pointer.

We solved this using the paging hardware to map all addresses that differ only in their tag bits to the same memory. This way, unmodified binary libraries can use tagged pointers, and instrumented code avoids the cost of clearing the tag too.

As shown in Figure 17(c), we use 5 bits to encode the size, allowing objects up to 2^{32} bytes. In order to use the paging hardware, these 5 bits have to come from the 43 bits supported by the operating system, thus leaving 38 bits of address space for programs.

With 5 address bits used for the bounds, we need to map 32 different address regions to the same memory. We implemented this entirely in user space using the `CreateFileMapping` and `MapViewOfFileEx` Windows API functions to replace the process image, stack, and heap with a file backed by the system paging file and mapped at 32 different locations in the process address space.

We use the 5 bits effectively ignored by the hardware to store the size of memory allocations. For heap allocations, our `malloc`-style functions set the tags for pointers they return. For locals and globals, we instrument the address taking operator "&" to properly tag the resulting pointer. We store the bit complement of the size logarithm enabling interoperability with untagged pointers by interpreting their zero bit pattern as all bits set (representing a maximal allocation of 2^{32}).

```
extract     {  mov rax, buf
bounds      {  shr rax, 26h
            {  xor rax, 1fh

pointer     {
arithmetic  {      char *p = buf[i];

            {  mov rbx, buf
bounds      {  xor rbx, p
check       {  shr rbx, al
            {  jz ok
            {    p = slowPath(buf, p)
            {  ok:
```

Figure 18: AMD64 code sequence inserted to check unsafe arithmetic with tagged pointers.

With the bounds encoded in pointers, there is no need for a memory lookup to check pointer arithmetic. Figure 18 shows the AMD64 code sequence for checking pointer arithmetic using a tagged pointer. First, we extract the encoded bounds from the source pointer by right shifting a copy to bring the tag to the bottom 8 bits of the register and xoring them with the value `0x1f` to recover the size logarithm by inverting the bottom 5 bits. Then we check that the result of the arithmetic is within bounds by xoring the source and result pointers, shifting the result by the tag stored in `al`, and checking for zero.

Similar to the table-based implementation of Section 3, out-of-bounds pointers trigger a bounds error to simplify the common case. To cause this, we zero the bits that were used to hold the size and save them using 5 more bits in the pointer, as shown in Figure 17(d).

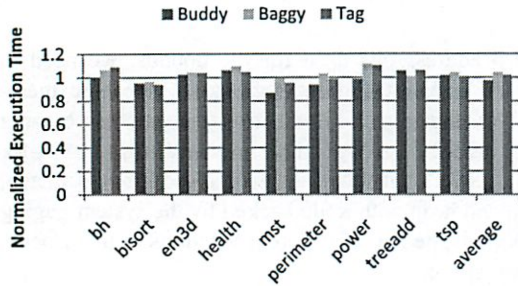


Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

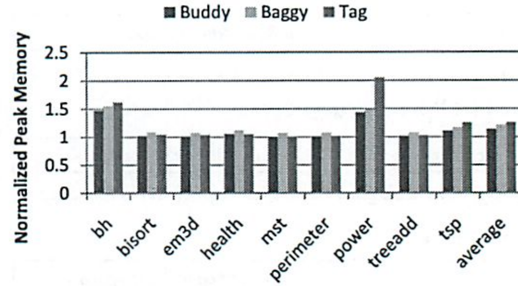


Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

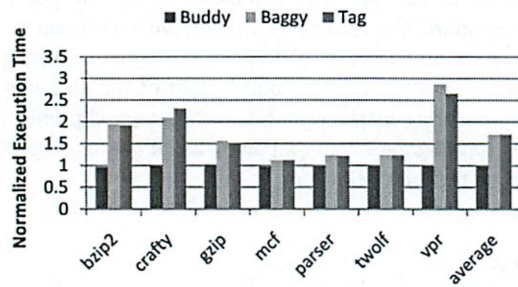


Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

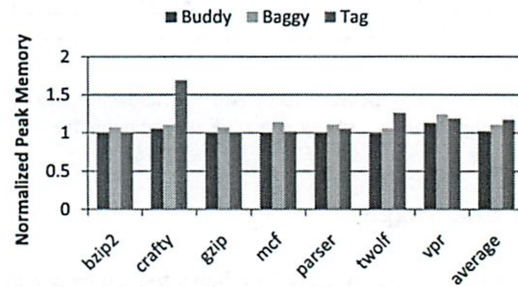


Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

5.2 Out-Of-Bounds Offset

The spare bits can also store an offset that allows us to adjust an out-of-bounds pointer to recover the address of its referent object. We can use 13 bits for this offset, as shown in Figure 17(d). These bits can count slot or even allocation size multiples, increasing the supported out-of-bounds range to at least 2^{16} bytes above or below an allocation.

This technique does not depend on size tagging and can be used with a table instead. When looking up a pointer in the table, however, the top bits have to be masked off.

5.3 Evaluation

We evaluated baggy bounds checking on AMD64 using the subset of benchmarks from Section 4.1 that run unmodified on 64 bits. We measured the system using a contiguous array against the system using tagged pointers (Baggy and Tag in the figure legends respectively). We also measured the overhead using the buddy allocator only.

The multiple memory mappings complicated measuring memory use because Windows counts shared memory multiple times in peak memory reports. To overcome this, we measured memory use without actually tagging the pointers, to avoid touching more than one address for the same memory, but with the memory mappings in place to account for at least the top level memory management overheads.

Figures 19 and 20 show the time overhead. The average using a table on 64-bits is 4% for Olden and 72% for SPECINT 2000—close to the 32-bit results of Section 3. Figures 21 and 22 show the space overhead. The average using a table is 21% for Olden and 11% for SPECINT 2000. Olden's space overhead is higher than the 32-bit version; unlike the 32-bit case, the buddy allocator contributes to this overhead by 14% on average.

Tagged pointers are 1–2% faster on average than the table, and use about 5% less memory for most benchmarks, except a few ones such as `power` and `crafty`. These exceptions are because our prototype does not map pages to different addresses on demand, but instead maps 32 30-bit regions of virtual address space on program startup. Hence the fixed overhead is notable for these benchmarks because their absolute memory usage is low.

While we successfully implemented mapping multiple views entirely in user-space, a robust implementation would probably require kernel mode support. We feel

that the gains are too small to justify the complexity. However, using the spare bits to store an out-of-bounds offset is a good solution for tracking out-of-bounds pointers when using the referent object approach of Jones and Kelly [19].

6 Related Work

Many techniques have been proposed to detect memory errors in C programs. Static analysis techniques, e.g., [33, 21, 7], can detect defects before software ships and they do not introduce runtime overhead, but they can miss defects and raise false alarms.

Since static techniques do not remove all defects, they have been complemented with dynamic techniques. Debugging tools such as Purify [17] and Annelid [25] can find memory errors during testing. While these tools can be used without source code, they typically slowdown applications by a factor of 10 or more. Some dynamic techniques detect specific errors such as stack overflows [13, 16, 32] or format string exploits [12]; they have low overhead but they cannot detect all spatial memory errors. Techniques such as control-flow integrity [20, 1] or taint tracking (e.g. [10, 26, 11, 35]) detect broad classes of errors, but they do not provide general protection from spatial memory errors.

Some systems provide probabilistic protection from memory errors [5]. In particular, DieHard [4] increases heap allocation sizes by a random amount to make more out-of-bounds errors benign at a low performance cost. Our system also increases the allocation size but enforces the allocation bounds to prevent errors and also protects stack-allocated objects in addition to heap-allocated ones.

Several systems prevent all spatial memory errors in C programs. Systems such as SafeC [3], CCured [24], Cyclone [18], and the technique in Xu *et al.* [36] associate bounds information with each pointer. CCured [24] and Cyclone [18] are memory safe dialects of C. They extend the pointer representation with bounds information, i.e., they use a fat pointer representation, but this changes memory layout and breaks binary compatibility. Moreover, they require a significant effort to port applications to the safe dialects. For example, CCured required changing 1287 out of 6000 lines of code for the Olden benchmarks [15], and an average of 10% of the lines of code have to be changed when porting programs from C to Cyclone [34]. CCured has 28% average runtime overhead for the Olden benchmarks, which is significantly higher than the baggy bounds overhead. Xu *et al.* [36] track pointers to detect spatial errors as well as temporal errors with additional overhead, thus their space overhead is proportional to the number of pointers. The average time overhead for spatial protection on the benchmarks we overlap is 73% versus 16% for baggy bounds with a space overhead of 273% versus 4%.

Other systems map any memory address within an allocated object to the bounds information for the object. Jones and Kelly [19] developed a backwards compatible bounds checking solution that uses a splay tree to map addresses to bounds. The splay tree is updated on allocation and deallocation, and operations on pointers are instrumented to lookup the bounds using an in-bounds pointer. The advantage over previous approaches using fat pointers is interoperability with code that was compiled without instrumentation. They increase the allocation size to support legal out-of-bounds pointers one byte beyond the object size. Baggy bounds checking offers similar interoperability with less time and space overhead, which we evaluated by using their implementation of splay trees with our system. CRED [30] improves on the solution of Jones and Kelly by adding support for tracking out-of-bounds pointers and making sure that they are never dereferenced unless they are brought within bounds again. Real programs often violate the C standard and contain such out-of-bounds pointers that may be saved to data structures. The performance overhead for programs that do not have out-of-bounds pointers is similar to Jones and Kelly if the same level of runtime checking is used, but the authors recommend only checking strings to lower the overhead to acceptable levels. For programs that do contain such out-of-bounds pointers the cost of tracking them includes scanning a hash-table on every dereference to remove entries for out-of-bounds pointers. Our solution is more efficient, and we propose ways to track common cases of out-of-bounds pointers that avoid using an additional data structure.

The fastest previous technique for bounds checking by Dhurjati *et al.* [15] is more than two times slower than our prototype. It uses inter-procedural data structure analysis to partition allocations into pools statically and uses a separate splay tree for each pool. They can avoid inserting some objects in the splay tree when the analysis finds that a pool is size-homogeneous. This should significantly reduce the memory usage of the splay tree compared to previous solutions, but unfortunately they do not report memory overheads. This work also optimizes the handling of out-of-bounds pointers in CRED [30] by relying on hardware memory protection to detect the dereference of out-of-bounds pointers.

The latest proposal, SoftBound [23], tracks bounds for each pointer to achieve sub-object protection. Sub-object

protection, however, may introduce compatibility problems with code using pointer arithmetic to traverse structures. SoftBound maintains interoperability by storing bounds in a hash table or a large contiguous array. Storing bounds for each pointer can lead to a worst case memory footprint as high as 300% for the hash-table version or 200% for the contiguous array. The average space overhead across Olden and a subset of SPECINT and SPECFP is 87% using a hash-table and 64% for the contiguous array, and the average runtime overhead for checking both reads and writes is 93% for the hash table and 67% for the contiguous array. Our average space overhead over Olden and SPECINT is 7.5% with an average time overhead of 32%.

Other approaches associate different kinds of metadata with memory regions to enforce safety properties. The technique in [37] detects some invalid pointers dereferences by marking all writable memory regions and preventing writes to non-writable memory; it reports an average runtime overhead of 97%. DFI [8] computes reaching definitions statically and enforces them at runtime. DFI has an average overhead of 104% on the SPEC benchmarks. WIT [2] computes the approximate set of objects written by each instruction and dynamically prevents writes to objects not in the set. WIT does not protect from invalid reads, and is subject to the precision of a points-to analysis when detecting some out-of-bounds errors. On the other hand, WIT can detect accesses to deallocated/unallocated objects and some accesses through dangling pointers to re-allocated objects in different analysis sets. WIT is six times faster than *baggy bounds checking* for SPECINT 2000, so it is also an attractive point in the error coverage/performance design space.

7 Limitations and Future Work

Our system shares some limitations with other solutions based on the referent object approach. Arithmetic on integers holding addresses is unchecked, casting an integer that holds an out-of-bounds address back to a pointer or passing an out-of-bounds pointer to unchecked code will break the program, and custom memory allocators reduce protection.

Our system does not address temporal memory safety violations (accesses through "dangling pointers" to re-allocated memory). Conservative garbage collection for C [6] is one way to address these but introduces its own compatibility issues and unpredictable overheads.

Our approach cannot protect from memory errors in sub-objects such as structure fields. To offer such protection, a system must track the bounds of each pointer [23] and risk false alarms for some legal programs that use pointers to navigate across structure fields.

In Section 4 we found two programs using out-of-bounds pointers beyond the $slot_size/2$ bytes supported on 32-bits and one beyond the 2^{16} bytes supported on 64-bits. Unfortunately the real applications built in Section 4.3 were limited to software we could readily port to the Windows toolchain; wide use will likely encounter occasional problems with out-of-bounds pointers, especially on 32-bit systems. We plan to extended our system to support all out-of-bounds pointers using the data structure from [30], but take advantage of the more efficient mechanisms we described for the common cases. To solve the delayed deallocation problem discussed in Section 6 and deallocate entries as soon as the out-of-bounds pointer is deallocated, we can track out-of-bounds pointers using the pointer's address instead of the pointer's referent object's address. (Similar to the approach [23] takes for all pointers.) To optimize scanning this data structure on every deallocation we can use an array with an entry for every few memory pages. A single memory read from this array on deallocation (e.g. on function exit) is sufficient to confirm the data structure has no entries for a memory range. This is the common case since most out-of-bounds pointers are handled by the other mechanisms we described in this paper.

Our prototype uses a simple intra-procedural analysis to find safe operations and does not eliminate redundant checks. We expect that integrating state of the art analyses to reduce the number of checks will further improve performance.

Finally, our approach tolerates harmless bound violations making it less suitable for debugging than slower techniques that can uncover these errors. On the other hand, being faster makes it more suitable for production runs, and tolerating faults in production runs may be desired [29].

8 Conclusions

Attacks that exploit out-of-bounds errors in C and C++ continue to be a serious security problem. We presented *baggy bounds checking*, a backwards-compatible bounds checking technique that implements efficient bounds checks. It improves the performance of bounds checks by checking allocation bounds instead of object bounds and by using a binary buddy allocator to constrain the size and alignment of allocations to powers of 2. These constraints enable a concise representation for allocation bounds and let *baggy bounds checking* store this infor-

mation in an array that can be looked up and maintained efficiently. Our experiments show that replacing a splay tree, which was used to store bounds information in previous systems, by our array reduces time overhead by an order of magnitude without increasing space overhead.

We believe *baggy bounds checking* can be used in practice to harden security-critical applications because it has low overhead, it works on unmodified C and C++ programs, and it preserves binary compatibility with uninstrumented libraries. For example, we were able to compile the Apache Web server with *baggy bounds checking* and the throughput of the hardened version of the server decreases by only 8% relative to an unistrumented version.

Acknowledgments

We thank our shepherd R. Sekar, the anonymous reviewers, and the members of the Networks and Operating Systems group at Cambridge University for comments that helped improve this paper. We also thank Dinakar Dhurjati and Vikram Adve for their communication.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[4] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2006.

[5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software Practice & Experience*, 1988.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software Practice & Experience*, 2000.

[8] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[9] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[10] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Can we contain Internet worms? In *Proceedings of the Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[11] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.

[12] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[14] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[16] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/main.html`.

[17] Reed Hasting and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the USENIX Annual Conference*, 2002.

[19] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)*, 1997.

[20] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[21] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[22] Microsoft. Phoenix compiler framework. `http://connect.microsoft.com/Phoenix`.

[23] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[24] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[25] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.

[26] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*, 2005.

[27] NullLogic. Null HTTPd. `http://nullwebmail.sourceforge.net/httpd`.

[28] OpenSSL Toolkit. `http://www.openssl.org`.

[29] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[30] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[31] The Apache Software Foundation. The Apache HTTP Server Project. `http://httpd.apache.org`.

[32] Vendicator. StackShield. `http://www.angelfire.com/sk/stackshield`.

[33] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS)*, 2000.

[34] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, 2003.

[35] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[36] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, 2004.

[37] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

6.858: Computer Systems Security

Fall **2012**

Paper Reading Questions

For each paper, your assignment is two-fold:

- Submit your answer for each lecture's paper question via the submission web site, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Suppose `slot_size` is set to 16 bytes. Consider the following code snippet:

```
char *p = malloc(256);
char *q = p + 256;
char ch = *q;
```

Explain whether or not baggy bounds checking will raise an exception at the dereference of `q`.

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Paper Qu

9/[illegible]

slot_size = 16 bytes

allocate *p 256 bytes

↑ pointer to p

then pointer to qu is p + 256

and i put character in as q

∟ basically assess it

So we should be outside allocation

so we are in illegal territory

We never talked about basic buffer overflows...

Pointers review

pointer = address of start of block of memory

pass around instead of copying

(2)

32 bits = 4 bytes

↑ 0 or 1

↑ 8 bits

2^8 options = 256

ie one alpha numeric character

00 to FF in Hex

One hex is

$2^x = 16$

$\log_2(16) = 4$ bits

Char str_a[20] ← array of 20 char element arrays

Char *pointer ← pointer for char array

↑ * points to data of that type!

Pointer = str_a

↑ set pointer to start array

(3)

* Memory address is stored in pointer

~~*0x2ef~~

Printing

Normal → resolves pointe

& address of → prints value directly in pointer
└ the memory address

Or prints the memory address of the normal variable

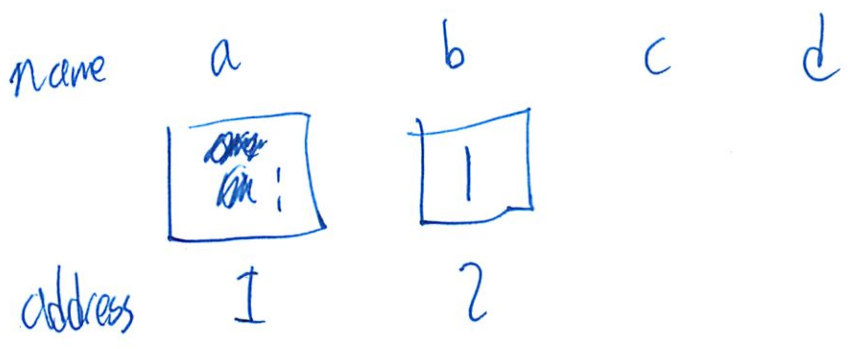
Dereference Operator

Returns data found in address pointer is pointing to - insted of address itself

kinda like a backwards

So ch is the character at p + 256

(4)



char a = ";"

Char *b = a

print b
└ 1 ← ~~see I think this should be 1~~
~~I think it is~~
Earlier example is clear

print *b
└ ;

print &b
└ 2

(I need a test env to run this stuff)

(5)

Woot! set up VIM

? single vs / double quote?

Getting it to print!

a[0] = 1 char array

&a seemed to be same as a

Tests not really working

What is the difference?

pointer = a

pointer = &a

? exactly finding in other book

L seems to make more sense

(6)

(R) So ~~char~~ str_a is an pointer to 1st el in array

^Jwang told me
1 hr of struggle
How do you scalably answer that qu

Or just put one line in a box!

Arrays = buffer

Question? 9/10

So allocate 256

So *p = 0
*q = 0 + 256
ch = deref *q

So should be out of bounds
But is it?
Slot = 16

(7)

~~Sh~~ Stores $e = \log_2(size)$

$\log_2(256)$

Which is ~~$8^2 = 256$~~

$2^8 = 256$

So allocation size is

size = 1 << 8

↑ bit shift left

= 256

? didn't do anything

base = p & ~(size - 1)

↑? not

Yes

? what is AND Not mean here

Look at bit wise

0000

7 is 0111
8 4 2 1

So Not 1000

and 0000
1000
= ———
1000 = 8 = base

? So what is that
└ the mask?

I should do a diff example some time

Oh that was previous!

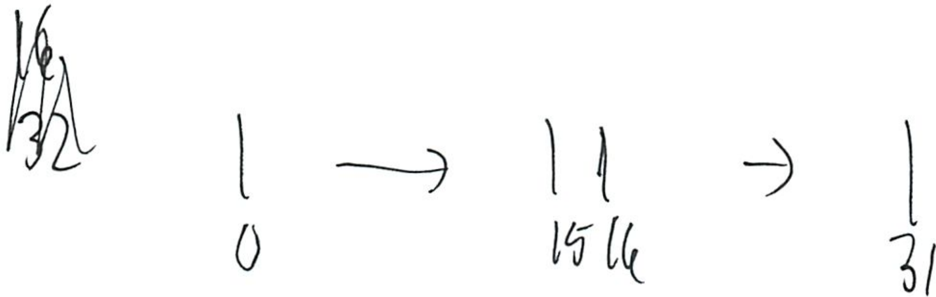
Partion memory into 16 bytes
entry for each slot 1/16 overhead
So $\log_2(16) = 4$
right shift address
add to constant table base

9

Bit each entry single byte
Slot size is additional

entry for each slot
∴ So 1 byte per slot

So every 16 bytes



So we have 16 slots for our memory
∴ We are right at edge - so ok∴
Can pad within

Lower + upper bounds
Check if result is 0
+ ?how

(10)

?so true since at boundary

└ will put that since out of time

?? too easy ???

— ?Plus - it should just work

9/10

Paper Question 1

Michael Plasmeier

256 bytes of memory is allocated. The slot size is 16 bytes. Since 256%16 =0, the boundaries line up and the baggy boundary checking will work when we attempt to dereference outside of the allocated memory.

TA thought this was right
└ un confirmed

6.858

4/10

L2 Buffer Overflows

Web Server Code

```
void read-req() {
    char buf[128];
    int i;
    gets(buf);
    :
    :
}
```

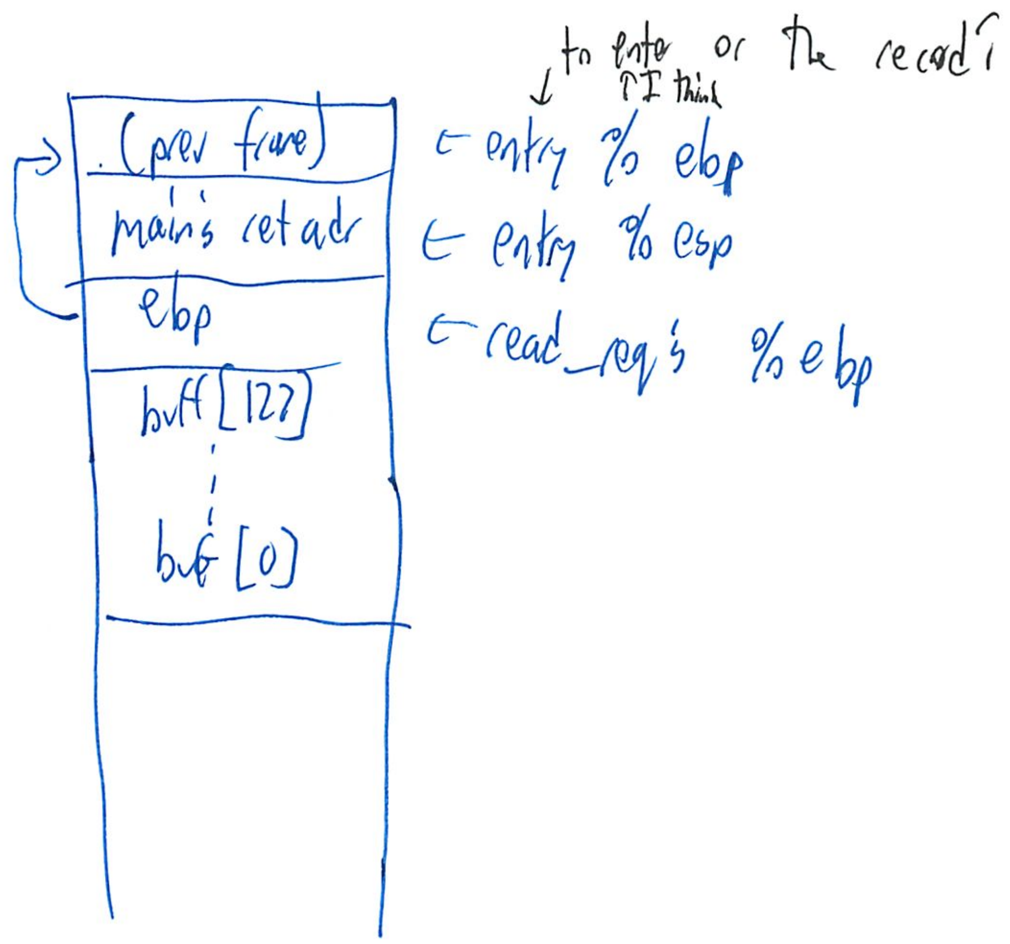
Caller → ~~call~~ call read-req

read_req:

```
push %ebp
mov  %esp → %ebp
sub  168, %esp
.... (call gets, etc)
mov  %ebp → %esp
pop  %ebp
ret
```

②

ebp = top of current fn on stack
for debugging



So then when returns (ret)
if return address is wrong
will jump there

So can set address to 128 byte buffer
that buffer had some code to run Linux shell
"exec"

(3)

Weak threat Model

Virtual Memory actually makes things eaiser
- Consistent from machine to machine

Why do these problems keep showing up?
Was worse earlier
People didn't write in a security-aware mode

Plus its hard
Some bugs can be fairly subtile

All sorts of errors can be exploitable
like reusing pointers
any sort of memory management issues

(4)

How to prevent?

- no bugs ← write better code
  - don't use gets, strcopy, sprintf
  - no info about length
  - instead fgets, strncpy, snprintf
  - but this is hard!
    - each has its weird things
    - will it null term string?
- Or mitigate issue
  - help programmers find bugs
  - Compilers
  - analysis tools
    - fuzzers - send arbitrary inputs till crashes
- Or memory safe language
  - Java, Python, C#
  - but lots of stuff written in C for historical or performance reasons

Mitigating

Attacker must
- Take control of program counter i.e. by overwriting return address
- Interesting code to execute

(5)

Baggy Bounds Checking

People don't really use in progress

Actual Mitigation Strategies

#1 ~~Instead~~ use Stack Canaries

- ~~check~~ some memory
- Check if overwritten before ~~doing~~ continuing
- if is → crash
  - better than ~~one~~ arbitrary code

- called gec in gcc

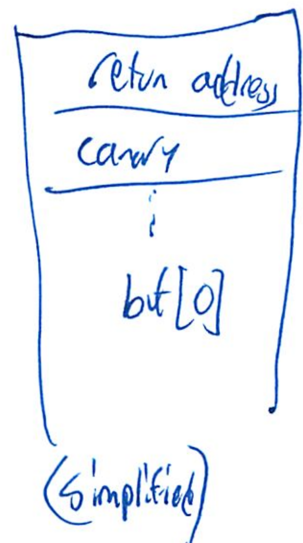
So in code

push canary

⋮ code

pop canary

Check



(6)

So what is it?

"Terminator" canary 0, CR, LF, EOF

many fns are string functions

They stop when they see these

ie gets fn will stop when see this

So either attacker writes this code

+ gets stop

or canary check fails

but only for certain string code

Random Canary

Generate random variable

Make sure still there

But need good randomness

w/o much overhead

7

? But where to store?

└ lots of attacks ~~allow~~ 1st look to read arbitrary memory to read canary

or write arbitrary memory

then buffer overflow

? read only - don't want to waste whole pg memory

but fork means you know the canary for other problems forked from same place

none of these are perfect
programs often use multiple

also doesn't protect function pointers

```
void (*f) (void);
gets(...)
f()
```

⑧

Microsoft compile tries to rearrange ~~the~~ stack
to minimize

#2 Bounds Checking

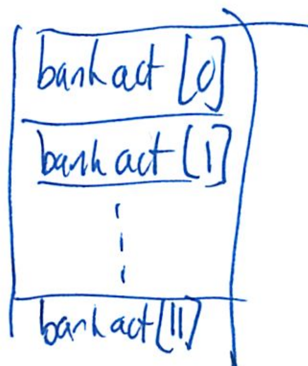
When read pointer → has pointer escaped
pointer no longer pointing outside original object

ahh so it saves allocated size
then start + size should = lenght

But what is out-of-bounds?

```
Struct bank act {
    Char name [128];
    int bal;
}
Struct bank act [12];
```

~~bank~~ ↳



9

What check?

- each variable
  - ie name, bal
- each object
  - ie bankact[0]
- entire ~~object~~ struct ← Baggy does 2

Baggy requires power of 2

Some false neg (missed attacks)

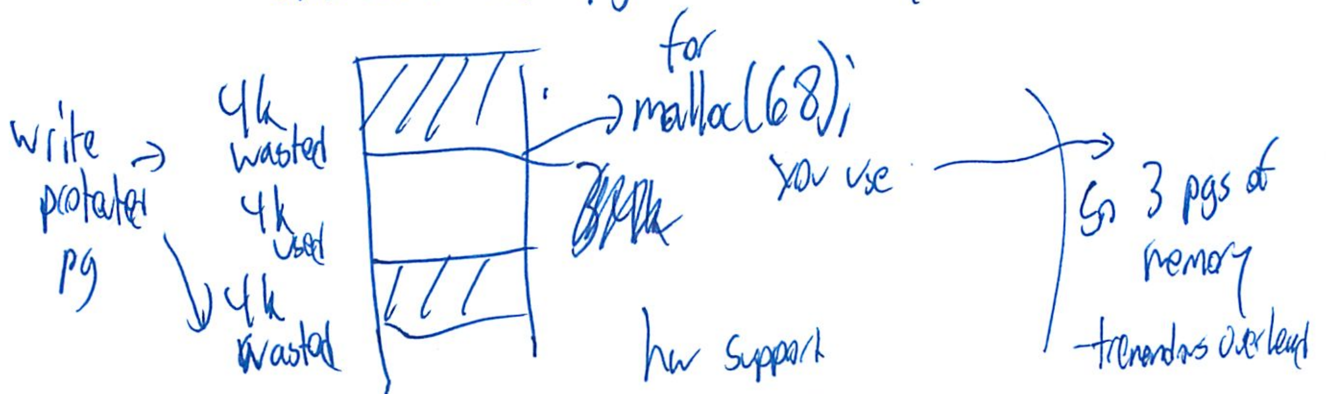
All bounds checking have these issues

Other Bounds Checking

- also prevent crashes

#1. Electric Fence

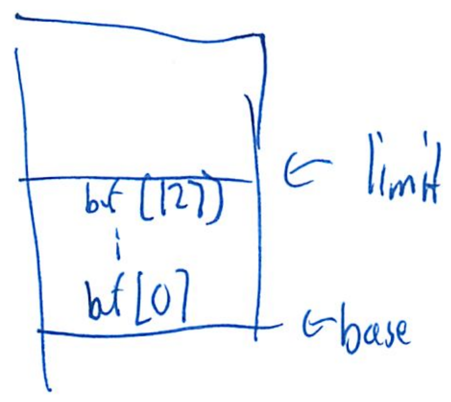
allocates sep pg for memory allocation



(10) must use compiler support to do this w/ less memory

need base and limit

does going forward + back drop out



#2 Fat pointer   every pointer is struct {
void *ptr
void *base
void *limit
}

each 12 bytes

2 places that matter
- pointer arithmatic
- dereference

```
char *p = malloc(n)
char *q = p+20
char c = *q
```

(11)

Compiler must see ~~base~~ + limit

~~ptrs~~ start location of alloc n

a pointer
- preserve base, limit
- adds 20 to ptr

dereference
- check that b/w base + limit

But not very compatible w/ how people write
C code
L people consider pointers small
or assume can cast point as int

So compat is why not popular

So instead → side data structure
Which is what Baggy Bounds checker
must impose on both - arimetric
- deref

; Or can we just interpose on deref

* Remember compiler is producing assembly

; Or just interpose on aritmatic

baggy ~~[illegible]~~ doesn't track deref
 but marks address so program will crash

Prof; very hard to do w/ arithmatic
 Since must know where pointer came from
 can't just track location in memory
 must know if we overflowed
 So must impose on pointer math
 this is ~~not~~ costly

(13)

Strawman Splay Tree

- L Some sort of lookup table
  from pointer to base and limit
  like fat pointer - but seperate

Open Issues When goes out of bounds -> must do something
What do on dereference

Dereference

Instead of modifying c = *q

Make sure pointer is out of bounds
L high bit set

legal 0x0000 1000
illegal 0x 8000 1000
~ usually kernal mem
so seg fault

Also faster

(14)

Storing

instead of Splay Tree
have a table

naive plan -> each byte of mem has a bit in table (right)

instead granularity of slot_size
ie 16 bytes

~~require~~
plus require each object to be power of 2
So just store the log
So 5 bits or 1 byte

example

```
char *p = malloc(0x2c);   ← 44 bytes
char *q = p + 0x3c
char *r = q + 0x10
char *s = r - 8;
char *t = s - 0x20;
```

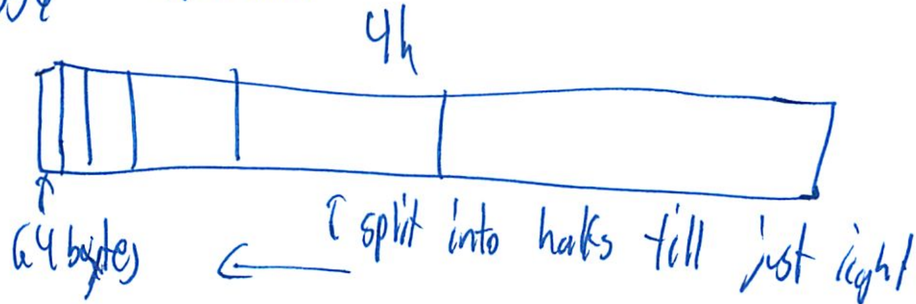
15

So the malloc(0x2c) 0x 0001040

↑ 44 bytes will round up to 64 - so power of 2

↑ address

~~buggy~~ buddy allocator



Then rest is left for other regions

when free it → it checks if buddy can be freed

char *q = p + 0x3c;

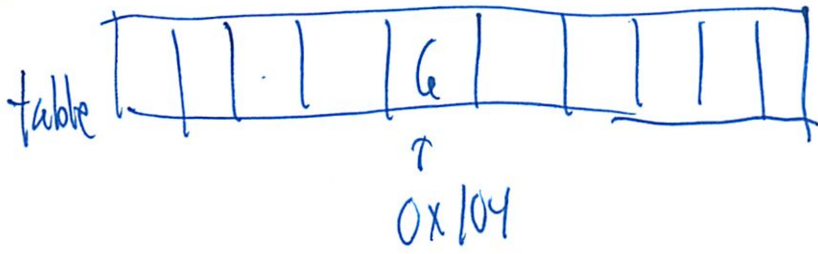
↑ will look up p

↑ compute q

↑ then check if same allocation

```
mov p %eax                  0x1040
shr %eax, 4 bits            0x00104   ← slot size division
mov table[%eax] → %el  | look up in table
```

(16)



then check if q and p in same object

Xor p, q

∟ is 0 if shifted by al

gives you bits that differ

~~Then~~

q is 0x 107c

So XOR is 0x 3c

bits that differ

* Can't differ at any bits greater than size of object *

When right shift by 6 bits (↑ size)

get 0 → so same

(17)

↓ 46 bytes

0x3c is larger than 0x2c allocated
but padded to 64 bytes

Then char *r = q + 0x10
r is larger than 64 bits

crossed allocation size boundry

So when xor 0xf0
When shift 6
its non 0

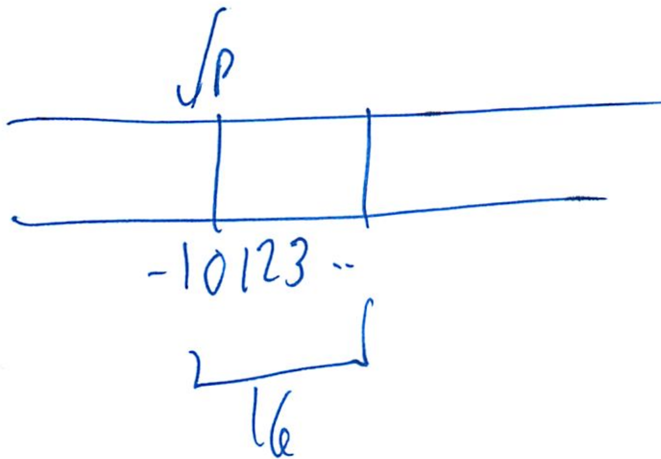
So set high order bit to 8
So program will crash
0x8000108c

False alarms are huge deal
why none of these adopted
people don't want to deal w/ this

(18)

Many kinda-legit out of bounds uses

char *p = a-1; — which they call "distasteful"



So index from 1, not 0

Or char *p = p+(a-b)

the shortcut

(p+a)-b

often worked
but technically illegal

(19)

char *s = r - 8   still out of bounds

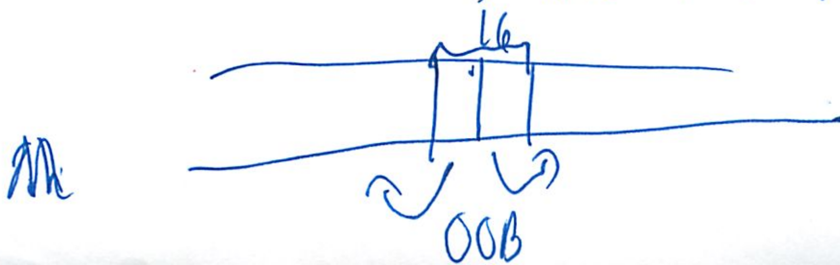
char *t = s - 0x20

but is finally out of bounds
so clear the high order bit
Now you can dereference it again!

table is ~6% of memory
more terrible

Can only get pointers over 8 bytes out of band
See high bit pointer   ? slotsize/2 restriction

if < slot, then from left   (? huh)
> slot, then from right



(20) Clarity

If goes out of bounds then comes back
by more than $\frac{slot}{2}$

it won't know how to fix it

(I know understand functionally ~~when~~ what breaks
but not why this limitation exists)

L2 9/10

```
Buffer overflows, memory safety
===============================

Some bugs in the paper:
  Figure 3, optimized bounds check should probably be
    (p^p') >> table[p >> log_of_slot_size] == 0
  Figures 5 and 18, pointer arithmetic code should probably be
    char *p = &buf[i];
  or
    char *p = buf + i;

Recall what's going on with buffer overflows (last lecture, 6.033, ..)
  Consider the following example code from, say, a web server:

    void read_req(void) {
      char buf[128];
      int i;
      gets(buf);
      .. parse buf ..
    }

  What does the compiler generate in terms of memory layout?
    x86 stack:
      Stack grows down.
      %esp points to the last (bottom-most) valid thing on the stack.
      %ebp points to the caller's %esp value.

                           +------------------+
        entry %ebp ----> | .. prev frame .. |
                         |                  |
                         |                  |
                         +------------------+
        entry %esp ----> |  return address  |
                         +------------------+
        new %ebp ------> |    saved %ebp    |
                         +------------------+
                         |     buf[127]     |
                         |       ...        |
                         |      buf[0]      |
                         +------------------+
        new %esp ------> |        i         |
                         +------------------+

    Caller's code (say, main):
      call read_req

    read_req's code:
      push    %ebp
      mov     %esp -> %ebp
      sub     168, %esp        # stack vars, etc
      ...
      mov     %ebp -> %esp
      pop     %ebp
      ret

  What's the threat model, policy?
    Assume that adversary can connect to web server, supply any inputs.
    Policy is a bit loose: only perform operations intended by programmer?
    In practice: don't want adversary to send spam, steal data, install bot.

  How does the adversary take advantage of this code?
    Supply long input, overwrite data on stack past buffer.
    Interesting bit of data: return address, gets used by 'ret'.
    Can set return address to the buffer itself, include some code in there.

  How does the adversary know the address of the buffer?
    What if one machine has twice as much memory?
    Luckily for adversary, virtual memory makes things more deterministic.
```

```
        For a given OS and program, addresses will often be the same.

      What happens if stack grows up, instead of down?
        Look at the stack frame for gets.

      What can the adversary do once they are executing code?
        Use any privileges of the process.
        Often leverage overflow to gain easier access into system.
          Originally on Unix, run shell /bin/sh (thus, "shell code").
        If the process is running as root or Administrator, can do anything.
        Even if not, can still send spam, read files (web server, database), ..
        Can attack other machines behind a firewall.

      Why would programmers write such code?
        Legacy code, wasn't exposed to the internet.
        Programmers were not thinking about security.
        Many standard functions used to be unsafe (strcpy, gets, sprintf).
        Even safe versions have gotchas (strncpy does not null-terminate).

      More generally, any memory errors can translate into a vulnerability.
        Using memory after it has been deallocated (use-after-free).
          If writing, overwrite new data structure, e.g. function ptr.
          If reading, might call a corrupted function pointer.
        Freeing the same memory twice (double-free).
          Might cause malloc to later return the same memory twice.
        Decrementing the stack ptr past the end of stack, into some other memory.
          [ http://www.invisiblethingslab.com/resources/misc-2010/xorg-large-memory-attacks.pdf ]
        Might not even need to overwrite a return address or function pointer.
          Can suffice to read sensitive data like an encryption key.
          Can suffice to change some bits (e.g. int isLoggedIn, int isRoot).

    Fixing buffer overflows, plan 1: avoid bugs in C code.
      Carefully check sizes of buffers, strings, arrays, etc.
      Use functions that take buffer sizes into account (strncpy, fgets, snprintf).
        gcc warns when a program uses gets() now.
        Other potentially dangerous functions are still widespread.
      Good: prevents problem in the first place.
      Bad: hard to ensure that code is bug-free, especially large existing code.

    Fixing buffer overflows, plan 2: help programmers find bugs.
      Works well in practice; will look at static analysis in later lectures.
      "Fuzzers" that supply random inputs can be effective for some kinds of bugs.
      Generally, hard to prove the absence of bugs, esp. for unsafe code like C.
      But, even partial analysis is useful (e.g. for Baggy bounds checking).

    Fixing buffer overflows, plan 3: use a memory-safe language (Java, C#, Python).
      Good: does prevent memory corruption errors.
        Except that low-level bindings still need to be correct.
        E.g. language runtime itself, or generic bindings like Python ctypes.
      Bad: still have a lot of legacy code in unsafe languages.
      Bad: might not perform as well as a fine-tuned C application?
        Used to be a bigger problem.
        Hardware & high-level languages are getting better.

    All 3 above approaches are effective and widely used, but not enough.
      Large/complicated legacy code written in C is still a big problem.
      Even newly written code in C/C++ can have memory errors.

    How to mitigate buffer overflows despite buggy code?
      Two things going on in a "traditional" buffer overflow:
        1: Adversary gains control over execution (program counter).
        2: Adversary executes some malicious code.
      What are the difficulties to these two steps?
        1: Requires overwriting a code pointer (which is later invoked).
           Common target is a return address using a buffer on the stack.
           Any memory error could potentially work, in practice.
           Function pointers, C++ vtables, exception handlers, ..
        2: Requires some interesting code in process's memory.
           Often easier than #1.
```

```
      Process already contains a lot of code.
      Process accepts inputs that adversary can supply.
      (But, adversary needs to find a predictable location.)

Mitigation approach 1: canaries (StackGuard, gcc's SSP, ..)
  Idea: OK to overwrite code ptr, as long as we catch it before invocation.
  One of the earlier systems: StackGuard
  Place a canary on the stack upon entry, check canary value before return.
    Usually requires source code; compiler inserts canary checks.
    Where is the canary on the stack diagram?
  Make the canary hard to forge:
    "Terminator canary": four bytes (0, CR, LF, -1)
      Idea: many C functions treat these characters as terminators.
      As a result, if canary matched, then further writes didn't happen.
    Random canary: much more common today (but, need good randomness!)
  What kinds of vulnerabilities will a stack canary not catch?
    Overwrites of function pointer variables before the canary.
    Heap object overflows (function pointers, C++ vtables).
    Overwrite a data pointer, then leverage it to do arbitrary mem writes.
      int *ptr = ...;
      char buf[128];
      gets(buf);
      *ptr = 5;
    Write directly to return address past the canary (for some buggy code).
  How could you trick the canary?
    Guess or obtain the canary value (if random)
      Maybe application leaks random values from its memory?
      Remove null terminator from a string, app will read later bytes.
    Overwrite the authentic canary value?
      If vulnerability allows arbitrary mem writes, can ovewrite it.

Mitigation approach 2: bounds checking.
  Overall goal: prevent pointer misuse by checking if pointers are in range.
    Sometimes hard to tell in C what's an overflow vs. what's legitimate.
      Suppose program allocates integer array, int x[1024];
              program also creates pointer, e.g. int *y = &x[107];
      Is it OK to increment y to access subsequent elements?
      If it's meant to be used like a string buffer, maybe yes.
      If it's meant to store bank account balances of many users, no.
    Gets even more difficult with mixed structs or unions.
    Usually, the plan is to at least enforce bounds between malloc objects.
      Or variables allocated by the compiler, either global or on stack.
    Often requires compiler changes (hard to do for unmodified, existing code).
  Electric fence: simple debugging tool from a while back.
    Align allocated memory objects with a guard page at end or beginning.
    Use page tables to ensure that accesses to guard page cause a crash.
    Reasonably effective as a debugging technique.
    Can prevent some buffer overflows for heap objects, but not for stack.
    Advantage: works without source code, no compiler changes.
      Need to be able to replace default malloc with efence's malloc.
    Problem: huge overhead (only 1 object per page, + guard pages).
  Fat pointers.
    Idea: modify a pointer representation to include bounds information.
      Requires a different compiler, access to source code.
    Usually called "fat pointer" because the pointer becomes much larger.
    E.g. simple scheme: 4-byte pointer, 4-byte obj base, 4-byte obj end.
    Compiler generates code to aborts if dereferencing pointer whose
      address is outside of its own base..end range.
    Problem 1: can be expensive to check all pointer dereferences.
    Problem 2: incompatible with a lot of existing code.
      Cannot pass fat pointer to unmodified library.
      Cannot use fat pointers in fixed-size data structures.
      Fat pointers are not atomic (some code assumes ptr writes are).
  Shadow data structures to keep track of bounds (Jones and Kelly, Baggy).
    Requires a significant amount of compiler support.
    For each allocated object, keep track of how big the object is.
      E.g. record the value passed to malloc: char *p = malloc(256);
      Or, for static variables, determined by compiler: char p[256];
    For each pointer, need to interpose on two operations:
```

```
    1. pointer arithmetic: char *q = p + 256;
    2. pointer dereferencing: char ch = *q;
    Why do we need to interpose on dereference?  (can we do just arithmetic?)
    Why do we need to interpose on arithmetic?  (can we do just dereference?)
  Need to ensure that out-of-bound pointers cannot touch other data.
  Challenge 1: looking up limits information for a regular pointer.
    Naive: hash table or interval tree for addrs.  Slow lookup.
    Naive: array w/ limit info for each memory addr.  Memory overhead.
  Challenge 2: causing out-of-bounds pointer dereferences to fail.
    Naive: interpose on every pointer dereference.  Expensive.
  What's the trick from the paper?
    1. Align, round up allocations to powers of 2: 5 bits of limit.
    2. Store limit info in a linear array: fast lookup.
    3. Allocate memory at slot granularity: fewer array entries.
    4. Use virtual memory system to prevent out-of-bound derefs.

    Why powers of 2?
      Can represent in a small amount of space (5 bits for 32-bit ptrs).
      Easy to check: base = p & ~(size-1).
      (p ^ p' >> table[p >> log_of_slot_size]) == 0
    What's a buddy allocator?
  What is the data structure that Baggy has in memory?
    Array at fixed virtual address, one byte per 16-byte mem "slot".
    Using virtual memory to allocate this array on-demand.
  Why don't these guys throw an error when arithmetic goes out-of-bounds?
    Applications simulate 1-indexed arrays.
    Applications want some value to represent one-past-the-end.
    Applications might compute OOB ptr but check later if it's OK to use.
    Applications may compute p+(a-b) as (p+a)-b.
  Example code:
    char *p = malloc(0x2c);    # 0x2c=44; 0x00001040 (suppose)
    char *q = p + 0x3c;        # 0x3c=60; 0x0000107c
    char *r = q + 0x10;        # 0x10=16; 0x8000108c
    char *s = r - 8;           #          0x80001084 [what happens if +8?]
    char *t = s - 0x20;        # 0x20=32; 0x00001064
  What does their pointer look like?
    Normal pointer for in-bounds values.
    High-bit-set for out-of-bounds (within half a slot).
      Typically, OS kernel lives in upper half, protects itself.
    Why half a slot for out-of-bounds?
  What happens when pointer arithmetic goes out-of-bounds?
    Instrumented code jumps to slow path, computes OOB pointer value.
    New pointer value points to "kernel memory", will cause crash.
    If OOB pointer converted to in-bounds pointer, high bit is cleared.
    In common case (all in bounds), no extra overhead.
  So what's the answer to the homework problem?
  Does Baggy instrument every memory address computation & access?
    Static analysis could prove some addr is always safe (no details).
    What's considered an "unsafe variable"?
      Address of variable taken, and not all uses can be proved safe.
    What does Baggy do with function call arguments?  Why?
      Cannot change, because x86 calling convention is fixed.
      Copy unsafe args to separate area, which is aligned & protected.
  How do they get binary compatibility with existing libraries?
    Normal pointers for in-bounds values.
    Set the bounds info to 31 (2^31 bound) for de-allocated memory.
    Solves problem of library code allocating its own memory.
  What can still go wrong with existing libraries?
    Can't detect out-of-bounds pointers generated in that code.
    Can't detect when OOB pointer passed into library goes in-bounds again.
  Why do they instrument strcpy and memcpy?  Do they need to?
  How does their 64-bit scheme work?
    Can get rid of the table storing bounds information, store in pointer.
    Can keep track of OOB pointers that go much further out-of-bounds (2^16).
  Does legitimate code always work with Baggy?
    Not quite; breaks in some cases.
    Unused out-of-bounds pointers.
    Temporary out-of-bounds pointers by more than slot_size/2.
    Conversion from pointer to integers and back.
```

```
      Passing out-of-bounds pointer into unchecked code.
    Can Baggy be bypassed to exploit a buffer overflow?
      Could exploit vulnerability in un-instrumented libraries.
      Could exploit temporal vulnerabilities (use-after-free).
      Mixed buffers and code pointers:

        struct {
            char buf[256];
            void (*f) (void);
        } my_type;
        struct my_type s;

      An overflow of s.buf will corrupt s.f, but not flag a bounds error.
      Would re-ordering f and buf help?
        Might break applications that depend on struct layout.
        Might not help if this is an array of (struct my_type)'s.
    What are the costs of bounds checking (e.g. Baggy)?
      Space overhead for limits information (fat pointer or extra table).
      Space overhead for extra padding memory used by buddy allocator (Baggy).
      CPU overheads for pointer arithmetic, dereferencing.
      False alarms!

Mitigation approach 3: non-executable memory (AMD's NX bit, Windows DEP, W^X, ..)
  Modern hardware allows specifying read, write, and execute perms for memory.
    R, W permissions were there a long time ago; execute is recent.
  Can mark the stack non-executable, so that adversary cannot run their code.
  More generally, some systems enforce "W^X", meaning all memory is either
    writable, or executable, but not both.  (Of course, OK to be neither.)
  Advantage: potentially works without any application changes.
  Shortcoming: harder to dynamically generate code (esp. with W^X).
    JITs like Java runtimes, Javascript engines, generate x86 on the fly.
    Can work around it, by first writing, then changing to executable.
    Java runtime used to mark all memory W+X (compatible, but not ideal).
  Can this still be exploited?
    Programs already contain a lot of code, might not need new code.
    Arc injection / return-to-libc attacks.
    Particularly useful functions: system, execl, unlink, strcpy, memcpy, ..
    General technique: "return-oriented programming"

Mitigation approach 4: randomized memory addresses (ASLR, stack randomization, ..)
  Make it difficult for adversary to guess a valid code pointer.
    Stack randomization: move stack to random locations, random offsets.
      Adversary doesn't know what to put for start of buf in return addr.
    Randomize entire address space (Address Space Layout Randomization):
      Rely on the fact that a lot of code is relocatable.
      Dynamic loader can choose random address for each library, program.
      Adversary doesn't know address of system(), etc.
  Can this still be exploited?
    Adversary might guess randomness.
      Especially on 32-bit machines, not a lot of random bits.
      32-bit address: 1 bit for kernel, 12 bit for page, at most 19 left.
      Most systems have more restrictions, leading to 8-16 random bits.
      More practical on 64-bit machines (easily 32 bits of randomness).
    Adversary might extract randomness.
      Programs might print a stack trace or error message w/ pointer.
      If adversary can run some code, they might get real addresses.
      Cute address leak in Flash's Dictionary (hash table):
        Get victim to visit your Flash-enabled page (e.g. buy an ad).
        Hash table internally computes hash value of keys.
        Hash value of integers is the integer.
        Hash value of object is its address.
        Iterating over a hash table is done in hash bucket order.
        Can compare address to integer, guess object address.
        Now, can exploit some code pointer overwrite and bypass ASLR.
    Adversary might not care exactly where to jump.
      "Heap spraying": fill memory w/ shellcode so that random jump is OK.
    Adversary might exploit some code that's not randomized (if exists).
  Some other interesting uses of randomization elsewhere.
    System call randomization (each process has its own system call numbers).
```

```
    Instruction set (perhaps machine or SQL language) randomization.

Are the mitigation techniques usable?  Should we use them?
  gcc and MSVC enable stack canaries by default.
  Linux, Windows include ASLR, NX by default.
  Fuzzing / fixing bugs / using memory-safe languages is common when possible.
  Bounds checking not as common (performance overheads, false alarms).
    Common thread in security tools: false alarms prevent adoption of tools.
    Often, 0 false alarms w/ some misses better than 0 misses w/ false alarms.
  Buffer overflows aren't #1 anymore; the web has "won" (SQL injection, XSS).
```

References:
  http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

OH 858 9/10

C Programming Lang Book
K & R

char str[20]

str[0] ← char
str ← char[20]
↳ ≈ char *
&str[0]

~~pointers are arrays~~

arrays are kinda pointers

sizeof (str) is array

a[b] if a is array is bth elem of array
a[b] if a is ptr is *(a + b)

---

integer + pointer

for char pointer - adding value
int 4 · value

(2)

(a + b) - c   weird quirk

a + (b - c)

Undefined behavior   spec does not say

Compiler can do whatever

ie overflow for signed int

if ((a + b) < b)

(a < 0)   ↙ compiler can do for complete

bad behavior

klang

Only valid pointers

point to some allocation

or one past it

↑ end of array

baggy sets flag since can go back in

(3)

So temp can be out of bounds
no compiler cares

baggy can't make assumption

---

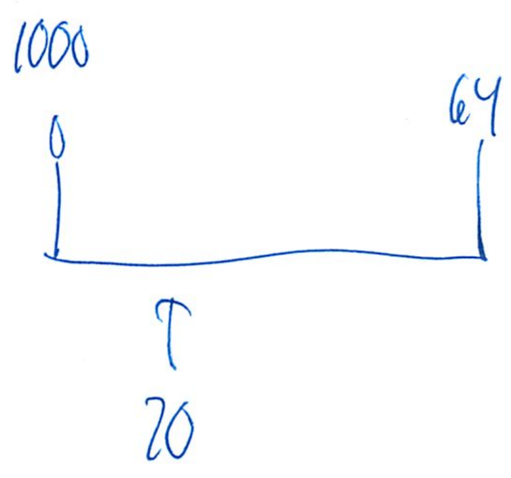
baggy can't recover if go more than 8 oob
legal for spec
but in pratice many have lots oob

fat pointer - still can recover base + limit
ie lenght
since printed in array

baggy - assume all allocations are powers of 2
every allocation alligned to power of 2
it is "aligned"

So 64 bytes
L must start at 0-64 64-128
not 4-68 etc

(4)



pointer to 1020

so what is size of object

Goals
1. know what obj points to
2. know its base + limit

fat pointer

allocate mem
- pointer = 1000
- base = 1000
- len = 64

add 20
- pointer = 1020
- base = 1000
- len = 64

add 80
- 1100
- 1000
- 64

(X) No - oob

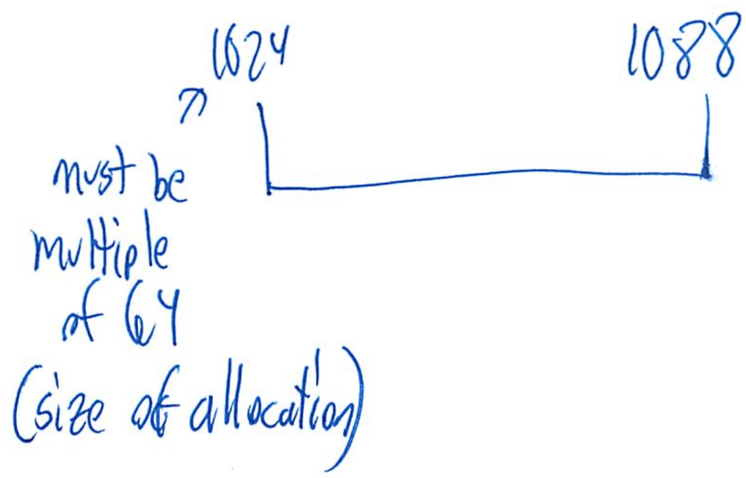
Only error if try to deref (in practice, technically illegal)

(5)

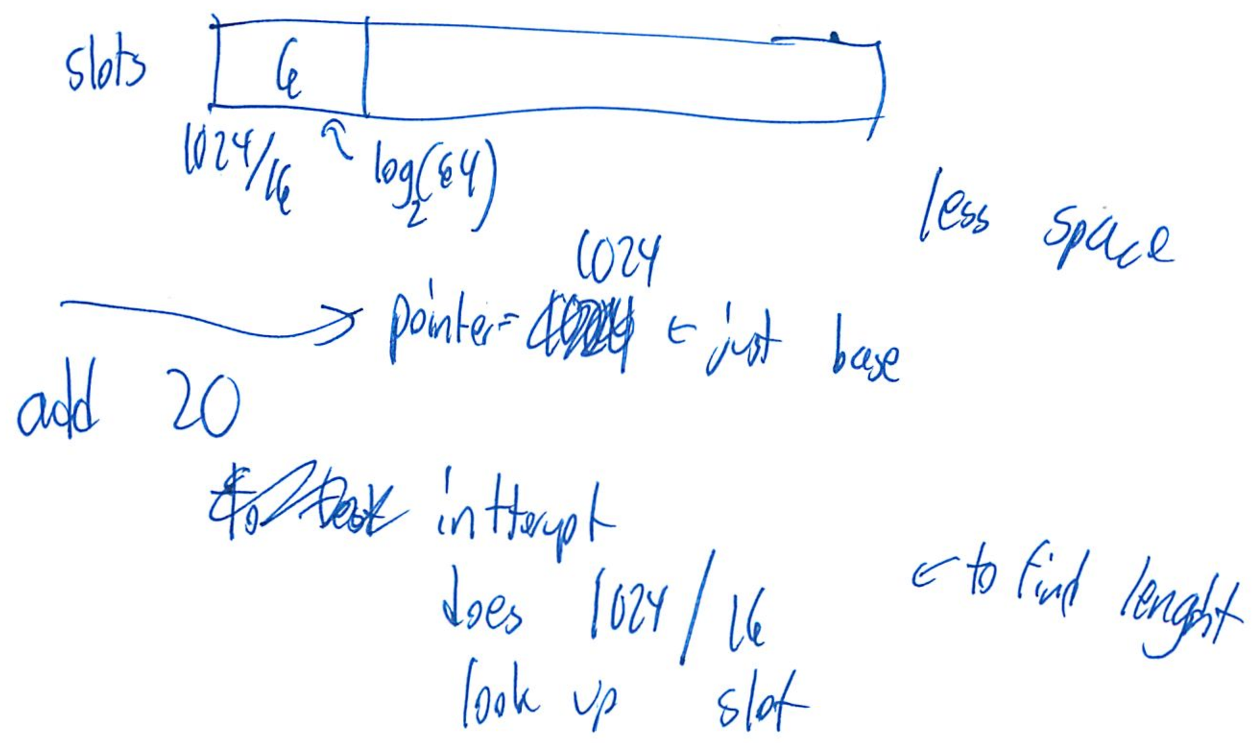
slot size = 16

baggy

allocate mem 64
round to power of 2
still 64



array slots to size allocation



(6)

Sees 6

$2^6 = 64$

is addition in bounds

64 - 20 = still positive

Now need base

know size

allocation is alligned to size

- mask off all bits before

2 cases

legal + adding ← doing

illegal + adding

know size and multiple of 64

So know next lowest or = multiple

So have base

Check → update pointer

(7)

illegal case

We can't go to next lowest
decided to split
instead could do ~~all~~ 16 ~~x~~ bits / direction

$$16 \times \frac{slot}{2}$$

Ans = trivially yes

Building Secure High-Performance Web Services with OKWS

Maxwell Krohn, *MIT Computer Science and AI Laboratory*
krohn@csail.mit.edu

Abstract

OKWS is a toolkit for building fast and secure Web services. It provides Web developers with a small set of tools that has proved powerful enough to build complex systems with limited effort. Despite its emphasis on security, OKWS shows performance improvements compared to popular systems: when servicing fully dynamic, non-disk-bound database workloads, OKWS's throughput and responsiveness exceed that of Apache 2 [3], Flash [23] and Haboob [44]. Experience with OKWS in a commercial deployment suggests it can reduce hardware and system management costs, while providing security guarantees absent in current systems.

1 Introduction

Most dynamic Web sites today maintain large server-side databases, to which their users have limited access via HTTP interfaces. Keeping this data hidden and correct is critical yet difficult. Indeed, headlines are replete with stories of the damage and embarrassment remote attackers can visit on large Web sites.

Most attacks against Web sites exploit weaknesses in popular Web servers or bugs in custom application-level logic. In practice, emphasis on rapid deployment and performance often comes at the expense of security.

Consider the following example: Web servers typically provide Web programmers with powerful and generic interfaces to underlying databases and rely on coarse-grained database-level permission systems for access control. Web servers also tend to package logically separate programs into one address space. If a particular Web site serves its *search* and *newsletter-subscribe* features from the same machine, a bug in the former might allow a malicious remote client to select all rows from a table of subscribers' email addresses. In general, anything from a buffer overrun to an unexpected escape sequence can expose private data to an attacker. Moreover, few practical isolation schemes exist aside from running different services on different machines. As a result, a flaw in one service can ripple through an entire system.

To plug the many security holes that plague existing Web servers, and to limit the severity of unforeseen problems, we introduce OKWS, the OK Web Server. Unlike typical Web servers, OKWS is specialized for dynamic content and is not well-suited to serving files from disk. It relies on existing Web servers, such as Flash [23] or Apache [3], to serve images and other static content. We argue (in Section 5.4) that this separation of static and dynamic content is natural and, moreover, contributes to security.

What OKWS does provide is a simple, powerful, and secure toolkit for building dynamic content pages (also known as *Web services*). OKWS enforces the natural principle of *least privilege* [27] so that those aspects of the system most vulnerable to attack are the least useful to attackers. Further, OKWS separates privileges so that the different components of the system distrust each other. Finally, the system distrusts the Web service developer, presuming him a sloppy programmer whose errors can cause significant damage. Though these principles are not novel, Web servers have not generally incorporated them.

Using OKWS to build Web services, we show that compromises among basic security principles, performance, and usability are unnecessary. To this effect, the next section surveys and categorizes attacks on Web servers, and Section 3 presents simple design principles that thwart them. Section 4 discusses OKWS's implementation of these principles, and Section 5 argues that the resulting system is practical for building large systems. Section 6 discusses the security achieved by the implementation, and Section 7 analyzes its performance, showing that OKWS's specialization for dynamic content helps it achieve better performance in simulated dynamic workloads than general purpose servers.

2 Brief Survey of Web Server Bugs

To justify our approach to dynamic Web server design, we briefly analyze the weaknesses of popular software packages. Our goal is to represent the range of bugs that have arisen in practice. Historically, attackers have exploited almost all aspects of conventional Web servers, from core components and scripting language exten-

sions to the scripts themselves. The conclusion we draw is that a better design—as opposed to a more correct implementation—is required to get better security properties.

In our survey, we focus on the Apache [3] server due to its popularity, but the types of problems discussed are common to all similar Web servers, including IBM WebSphere [14], Microsoft IIS [19] and Zeus [47].

2.1 Apache Core and Standard Modules

There have been hundreds of major bugs in Apache's core and in its standard modules. They fit into the following categories:

Unintended Data Disclosure. A class of bugs results from Apache delivering files over HTTP that are supposed to be private. For instance, a 2002 bug in Apache's `mod_dav` reveals source code of user-written scripts [42]. A recent discovery of leaked file descriptors allows remote users to access sensitive log information [7]. On Mac OS X operating systems, a local find-by-content indexing scheme creates a hidden yet world-readable file called `.FBCIndex` in each directory indexed. Versions of Apache released in 2002 expose this file to remote clients [41]. In all cases, attackers can use knowledge about local configuration and custom-written application code to mount more damaging attacks.

Buffer Overflows and Remote Code Execution. Buffer overflows in Apache and its many modules are common. Unchecked boundary conditions found recently in `mod_alias` and `mod_rewrite` regular expression code allow local attack [39]. In 2002, a common Apache deployment with OpenSSL had a critical bug in client key negotiation, allowing remote attackers to execute arbitrary code with the permissions of the Web server. The attacking code downloads, compiles and executes a program that seeks to infect other machines [36].

There have been less-sophisticated attacks that resulted in arbitrary remote code execution. Some Windows versions of Apache execute commands in URLs that follow pipe characters ('|'). A remote attacker can therefore issue the command of his choosing from an unmodified Web browser [40]. On MS-DOS-based systems, Apache failed to filter out special device names, allowing carefully-crafted HTTP POST requests to execute arbitrary code [43]. Other problems have occurred when site developers call Apache's `htdigest` utility from within CGI scripts to manage HTTP user authentication [6].

Denial of Service Attacks. Aside from TCP/IP-based DoS attacks, Apache has been vulnerable to a number of application-specific attacks. Apache versions released in 2003 failed to handle error conditions on certain "rarely used ports," and would stop servicing incoming connections as a result [38]. Another 2003 release allowed local configuration errors to result in infinite redirection loops [8]. In some versions of Apache, attackers could exhaust Apache's heap simply by sending a large sequence of linefeed characters [37].

2.2 Scripting Extensions to Apache

Apache's security worsens considerably when compiled with popular modules that enable dynamically-generated content such as PHP [25]. In the past two years alone, at least 13 critical buffer overruns have been found in the PHP core, some of which allowed attackers to remotely execute arbitrary code [9, 28]. In six other cases, faults in PHP allowed attackers to circumvent its application level *chroot*-like environment, called "Safe Mode." One vulnerability exposed `/etc/passwd` via `posix_getpwnam` [5]. Another allowed attackers to write PHP scripts to the server and then remotely execute them; this bug persisted across multiple releases of PHP intended as fixes [35].

Even if a correct implementation of PHP were possible, it would still provide Web programmers with ample opportunity to introduce their own vulnerabilities. A canonical example is that beginning PHP programmers fail to check for sequences such as "`..`" in user input and therefore inadvertently allow remote access to sensitive files higher up in the file system hierarchy (*e.g.*, `../../../etc/passwd`). Similarly, PHP scripts that embed unescaped user input inside SQL queries present openings for "SQL Injection." If a PHP programmer neglects to escape user input properly, a malicious user can turn a benign `SELECT` into a catastrophic `DELETE`.

The PHP manual does state that PHP scripts might be separated and run as different users to allow for privilege separation. In this case, however, PHP could not run as an Apache module, and the system would require a new PHP process forked for every incoming connection. This isolation strategy is at odds with performance.

3 Design

If we assume that bugs like the ones discussed above are inevitable when building a large system, the best remedy is to limit the effectiveness of attacks when they occur. This section presents four simple guidelines for protecting sensitive site data in the worst-case scenario, in which

an adversary remotely gains control of a Web server and can execute arbitrary commands with the Web server's privileges. We also present OKWS's design, which follows the four security guidelines without sacrificing performance.

Throughout, we assume a cluster of Web servers and database machines connected by a fast, firewalled LAN. Site data is cached at the Web servers and persistently stored on the database machines. The primary security goals are to prevent intrusion and to prevent unauthorized access to site data.

3.1 Practical Security Guidelines

(1) Server processes should be chrooted. After compromising a server process, most attackers will try to gain control over the entire server machine, possibly by installing "back doors," learning local passwords or private keys, or probing local configuration files for errors. At the very least, a compromised Web server should have no access to sensitive files or directories. Moreover, an OS-level jail ought to hide all `setuid` executables from the Web server, since many privilege escalation attacks require such files (examples include the *ptrace* and *bind* attacks mentioned in [17]). Privilege escalation is possible without `setuid` executables but requires OS-level bugs or race conditions that are typically rarer.

An adversary can still do damage without control of the Web server machine. The configuration files, source files, and binaries that correspond to the currently running Web server contain valuable hints about how to access important data. For instance, PHP scripts often include the username and plaintext password used to gain access to a MySQL database. OS-enforced policy ought to hide these files from running Web servers.

(2) Server processes should run as unprivileged users. A compromised process running as a privileged user can do significant damage even from within a *chroot*ed environment. It might bind to a well-known network port. It might also interfere with other system processes, especially those associated with the Web server: it can trace their system calls or send them signals.

(3) Server processes should have the minimal set of database access privileges necessary to perform their task. Separate processes should not have access to each other's databases. Moreover, if a Web server process requires only row-wise access to a table, an adversary who compromises it should not have the authority to perform operations over the entire table.

(4) A server architecture should separate independent functionality into independent processes. An adversary who compromises a Web server can examine its in-memory data structures, which might contain soft state used for user session management, or possibly secret tokens that the Web server uses to authenticate itself to its database. With control of a Web server process, an adversary might hijack an existing database connection or establish a new one with the authentication tokens it acquired. Though more unlikely, an attacker might also monitor and alter network traffic entering and exiting a compromised server.

The important security principle here is to limit the types of data that a single process can access. Site designers should partition their global set of site data into small, self-contained subsets, and their Web server ought to align its process boundaries with this partition.

If a Web server implements principles (1) through (4), and if there are no critical kernel bugs, an attacker cannot move from vulnerable to secure parts of the system. By incorporating these principles, a Web server design assumes that processes will be compromised and therefore prevents uncompromised processes from performing unsafe operations, even when extended by careless Web developers. For example, if a server architecture denies a successful attacker access to `/etc/passwd`, then a programmer cannot inadvertently expose this file to remote clients. Similarly, if a successful attacker cannot arbitrarily access underlying databases, then even a broken Web script cannot enable SQL injection attacks.

3.2 OKWS Design

We designed OKWS with these four principles in mind. OKWS provides Web developers with a set of libraries and helper processes so they can build Web services as independent, stand-alone processes, isolated almost entirely from the file system. The core libraries provide basic functionality for receiving HTTP requests, accessing data sources, composing an HTML-formatted response, responding to HTTP requests, and logging the results to disk. A process called OK launcher daemon, or *okld*, launches custom-built services and relaunches them should they crash. A process called OK dispatcher, or *okd*, routes incoming requests to appropriate Web services. A helper process called *pubd* provides Web services with limited read access to configuration files and HTML template files stored on the local disk. Finally, a dedicated logger daemon called *oklogd* writes log entries to disk. Figure 1 summarizes these relationships.

This architecture allows custom-built Web services to meet our stated design goals:

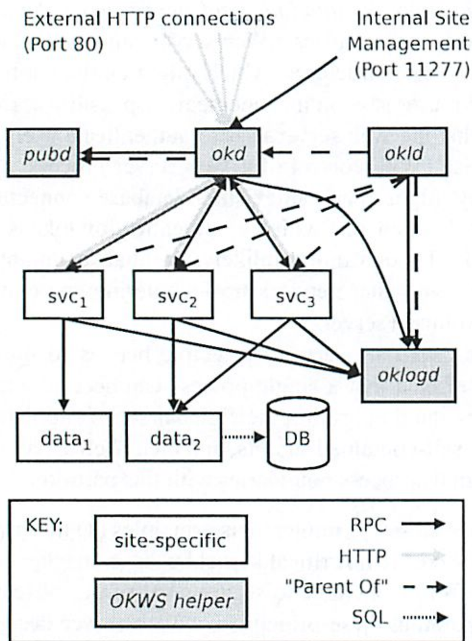


Figure 1: Block diagram of an OKWS site setup with three Web services (svc_1, svc_2, svc_3) and two data sources ($data_1, data_2$), one of which ($data_2$) is an OKWS database proxy.

(1) OKWS *chroots* all services to a remote jail directory. Within the jail, each process has just enough access privileges to read shared libraries upon startup and to dump core upon abnormal termination. The services otherwise never access the file system and lack the privileges to do so.

(2) Each service runs as a unique non-privileged user.

(3) OKWS interposes a structured RPC interface between the Web service and the database and uses a simple authentication mechanism to align the partition among database access methods with the partition among processes.

(4) Each Web service runs as a separate process. The next section justifies this choice.

3.3 Process Isolation

Unlike the other three principles, the fourth, of process isolation, implies a security and performance tradeoff since the most secure option—one Unix process per *external user*—would be problematic for performance. OKWS's approach to this tradeoff is to assign one Unix process per *service*; we now justify this selection.

Our approach is to view Web server architecture as a dependency graph, in which the nodes represent processes, services, users, and user state. An edge (a, b) denotes b's dependence on a, meaning an attacker's ability to compromise a implies an ability to compromise b. The crucial design decision is thus how to establish dependencies between the more abstract notions of services, users and user states, and the more concrete notion of a process.

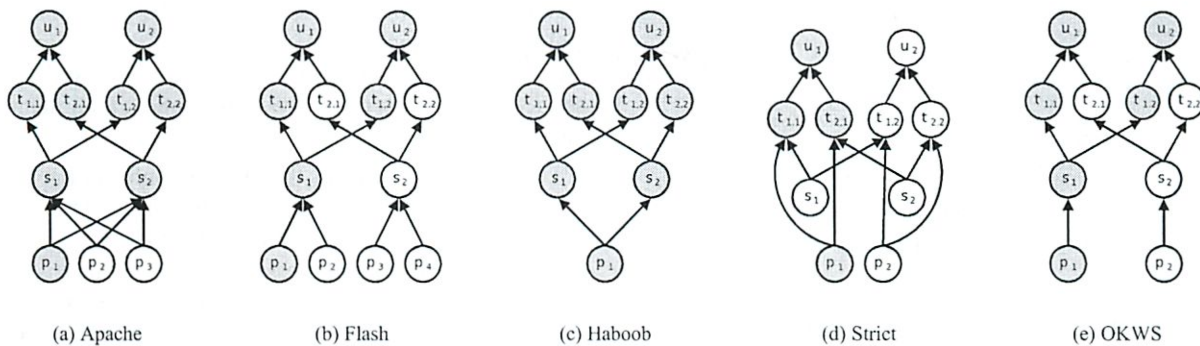
Let the set S represent a Web server's constituent services, and assume each service accesses a private pool of data. (Two application-level services that share data would thus be modelled by a single "service".) A set of users U interacts with these services, and the interaction between user u_j and service s_i involves a piece of state $t_{i,j}$. If an attacker can compromise a service s_i, he can compromise state $t_{i,j}$ for all j; thus $(s_i, t_{i,j})$ is a dependency for all j. Compromising state also compromises the corresponding user, so $(t_{i,j}, u_j)$ is also a dependency.

Let $P = \{p_1, \ldots, p_k\}$ be a Web server's pool of processes. The design decision of how to allocate processes reduces to where the nodes in P belong on the dependency graph. In the Apache architecture [3], each process p_i in the process pool can perform the role of any service s_j. Thus, dependencies (p_i, s_j) exist for all j. For Flash [3], each process in P is associated with a particular service: for each p_i, there exists s_j such that (p_i, s_j) is a dependency. The size of the process pool P is determined by the number of concurrent active HTTP sessions; each process p_i serves only one of these connections. Java-based systems like the Haboob Server [44] employ only one process; thus $P = \{p_1\}$, and dependencies (p_1, s_j) exist for all j.

Figures 2(a)-(c) depict graphs of Apache, Flash and Haboob hosting two services for two remote users. Assuming that the "dependence" relationship is transitive, and that an adversary can compromise p_1, the shaded nodes in the graph show all other vulnerable entities.

This picture assumes that the process of p_1 is equally vulnerable in the different architectures and that all architectures succeed equally in isolating different processes from each other. Neither of these assumptions is entirely true, and we will return to these issues in Section 6.2. What is clear from these graphs is that in the case of Flash, a compromise of p_1 does not affect states $t_{2,1}$ and $t_{2,2}$. For example, an attacker who gained access to u_i's search history ($t_{1,i}$) cannot access the contents of his inbox ($t_{2,i}$).

A more strict isolation strategy is shown in Figure 2(d). The architecture assigns a process p_i to each user u_i. If the attacker is a user u_i, he should only be able to compro-



(a) Apache (b) Flash (c) Haboob (d) Strict (e) OKWS

Figure 2: Dependency graphs for various Web server architectures.

mise his own process p_i, and will not have access to state belonging to other users u_j. The problem with this approach is that it does not scale well. A Web server would either need to fork a new process p_i for each incoming HTTP request or would have a large pool of mostly idle processes, one for each currently active user (of which there might be tens of thousands).

OKWS does not implement the strict isolation strategy but instead associates a single process with each individual service, shown in Figure 2(e). As a result OKWS achieves the same isolation properties as Flash but with a process pool whose size is independent of the number of concurrent HTTP connections.

4 Implementation

OKWS is a portable, event-based system, written in C++ with the SFS toolkit [18]. It has been successfully tested on Linux and FreeBSD. In OKWS, the different helper processes and site-specific services shown in Figure 1 communicate among themselves with SFS's implementation of Sun RPC [32]; they communicate with external Web clients via HTTP. Unlike other event-based servers [23, 44, 47], OKWS exposes the event architecture to Web developers.

To use OKWS, an administrator installs the helper binaries (*okld*, *okd*, *pubd* and *oklogd*) to a standard directory such as `/usr/local/sbin`, and installs the site-specific services to a runtime jail directory, such as `/var/okws/run`. The administrator should allocate two new UID/GID pairs for *okd* and *oklogd* and should also reserve a contiguous user and group ID space for "anonymous" services. Finally, administrators can tweak the master configuration file, `/etc/okws_config`. Table 1 summarizes the runtime configuration of OKWS.

4.1 okld

The root process in the OKWS system is *okld*—the launcher daemon. This process normally runs as superuser but can be run as a non-privileged user for testing or in other cases when the Web server need not bind to a privileged TCP port. When *okld* starts up, it reads the configuration file `/etc/okws_config` to determine the locations of the OKWS helper processes, the anonymous user ID range, which directories to use as jail directories, and which services to launch. Next, *okld* launches the logging daemon (*oklogd*) and the demultiplexing daemon (*okd*), and *chroots* into its runtime jail directory. It then launches all site-specific Web services. The steps for launching a single service are:

1. *okld* requests a new Unix socket connection from *oklogd*.

2. *okld* opens 2 socket pairs; one for HTTP connection forwarding, and one for RPC control messages.

3. *okld* calls `fork`.

4. In the child address space, *okld* picks a fresh UID/GID pair $(x.x)$, sets the new process's group list to $\{x\}$ and its UID to x. It then changes directories into `/cores/`x.

5. Still in the child address space, *okld* calls `execve`, launching the Web service. The new Web service process inherits three file descriptors: one for receiving forwarded HTTP connections, one for receiving RPC control messages, and one for RPC-based request logging. Some configuration parameters in `/etc/okws_config` are relevant to child services, and *okld* passes these to new children via the command line.

process	*chroot* jail	run directory	uid	gid
okld	/var/okws/run	/	root	wheel
pubd	/var/okws/htdocs	/	www	www
oklogd	/var/okws/log	/	oklogd	oklogd
okd	/var/okws/run	/	okd	okd
svc_1	/var/okws/run	/cores/51001	51001	51001
svc_2	/var/okws/run	/cores/51002	51002	51002
svc_3	/var/okws/run	/cores/51003	51003	51003

Table 1: An example configuration of OKWS. The entries in the "run directory" column are relative to "*chroot* jails".

6. In the parent address space, *okld* sends the server side of the sockets opened in Step 2 to *okd*.

Upon a service's first launch, *okld* assigns it a group and user ID chosen arbitrarily from the given range (*e.g.*, 51001-51080). The service gets those same user and group IDs in subsequent launches. It is important that no two services share a UID or GID, and *okld* ensures this invariant. The service executables themselves are owned by root, belong to the group with the anonymous GID x chosen in Step 4 and are set to mode `0410`.

These settings allow *okld* to call `execve` after `setuid` but disallow a service process from changing the mode of its corresponding binary. *okld* changes the ownerships and permissions of service executables at launch if they are not appropriately set. The directory used in Step 4 is the only one in the jailed file system to which the child service can write. If such a directory does not exist or has the wrong ownership or permissions, *okld* creates and configures it accordingly.

okld catches `SIGCHLD` when services die. Upon receiving a non-zero exit status, *okld* changes the owner and mode of any core files left behind, rendering them inaccessible to other OKWS processes. If a service exits uncleanly too many times in a given interval, *okld* will mark it broken and refuse to restart it. Otherwise, *okld* restarts dead services following the steps enumerated above.

4.2 okd

The *okd* process accepts incoming HTTP requests and demultiplexes them based on the "Request-URI" in their first lines. For example, the HTTP/1.1 standard [11] defines the first line of a GET request as:

GET /⟨*abs_path*⟩?⟨*query*⟩ HTTP/1.1

Upon receiving such a request, *okd* looks up a Web service corresponding to *abs_path* in its dispatch table. If successful, *okd* forwards the remote client's file descriptor to the requested service. If the lookup is successful but the service is marked "broken," *okd* sends an HTTP 500 error to the remote client. If the request did not match a known service, *okd* returns an HTTP 404 error. In typical settings, a small and fixed number of these services are available—on the order of 10. The set of available services is fixed once *okd* reads its configuration file at launch time.

Upon startup, *okd* reads the OKWS configuration file (`/etc/okws_config`) to construct its dispatch table. It inherits two file descriptors from *okld*: one for logging, and one for RPC control messages. *okd* then listens on the RPC channel for *okld* to send it the server side of the child services' HTTP and RPC connections (see Section 4.1, Step 6). *okd* receives one such pair for each service launched. The HTTP connection is the sink to which *okd* sends incoming HTTP requests from external clients after successful demultiplexing. Note that *okd* needs access to *oklogd* to log Error 404 and Error 500 messages.

okd also plays a role as a control message router for the child services. In addition to listening for HTTP connections on port 80, *okd* also listens for internal requests from an administration client. It services the two RPC calls: REPUB and RELAUNCH. A site maintainer should call the former to "activate" any changes she makes to HTML templates (see Section 4.4 for more details). Upon receiving a REPUB RPC, *okd* triggers a simple update protocol that propagates updated templates.

A site maintainer should issue a RELAUNCH RPC after updating a service's binary. Upon receiving a RELAUNCH RPC, *okd* simply sends an EOF to the relevant service on its control socket. When a Web service receives such an EOF, it finishes responding to all pending HTTP requests, flushes its logs, and then exits cleanly. The launcher daemon, *okld*, then catches the corresponding `SIGCHLD` and restarts the service.

4.3 oklogd

All services, along with *okd*, log their access and error activity to local files via *oklogd*—the logger daemon. Because these processes lack the privileges to write to the same log file directly, they instead send log updates over a local Unix domain socket. To reduce the total number of messages, services send log updates in batches. Services flush their log buffers as they become full and at regularly-scheduled intervals.

For security, *oklogd* runs as an unprivileged user in its own *chroot* environment. Thus, a compromised *okd* or Web service cannot maliciously overwrite or truncate log files; it would only have the ability to fill them with "noise."

4.4 pubd

Dynamic Web pages often contain large sections of static HTML code. In OKWS, such static blocks are called HTML "templates"; they are stored as regular files, can be shared by multiple services and can include each other in a manner similar to Server Side Includes [4].

OKWS services do not read templates directly from the file system. Rather, upon startup, the publishing daemon (*pubd*) parses and caches all required templates. It then ships parsed representations of the templates over RPC to other processes that require them. *pubd* runs as an unprivileged user, relegated to a jail directory that contains only public HTML templates. As a security precaution, *pubd* never updates the files it serves, and administrators should set its entire *chroot*ed directory tree read-only (perhaps, on those platforms that support it, by mounting a read-only *nullfs*).

5 OKWS In Practice

Though its design is motivated by security goals, OKWS provides developers with a convenient and powerful toolkit. Our experience suggests that OKWS is suitable for building and maintaining large commercial systems.

5.1 Web Services

A Web developer creates a new Web service as follows:

1. Extends two OKWS generic classes: one that corresponds to a long-lived service, and one that corresponds to an individual HTTP request. Implements the `init` method of the former and the `process` method of the latter.

2. Runs the source file through OKWS's preprocessor, which outputs C++ code.

3. Compiles this C++ code into an executable, and installs it in OKWS's service jail.

4. Adds the new service to `/etc/okws_config`.

5. Restarts OKWS to launch.

The resulting Web service is a single-threaded, event-driven process.

The OKWS core libraries handle the mundane mechanics of a service's life cycle and its connections to OKWS helper processes. At the initialization stage, a Web service establishes persistent connections to all needed databases. The connections last the lifetime of the service and are automatically reopened in the case of abnormal termination. Also at initialization, a Web service obtains static HTML templates and local configuration parameters from *pubd*. These data stay in memory until a message from *okd* over the RPC control channel signals that the Web service should refetch. In implementing the `init` method, the Web developer need only specify which database connections, templates and configuration files he requires.

The `process` method specifies the actions required for incoming HTTP requests. In formulating replies, a Web service typically accesses cached soft-state (such as user session information), database-resident hard state (such as inbox contents), HTML templates, and configuration parameters. Because a Web service is implemented as a single-threaded process, it does not require synchronization mechanisms when accessing these data sources. Its accesses to a database on behalf of all users are pipelined through a single asynchronous RPC channel. Similarly, its accesses to cached data are guaranteed to be atomic and can be achieved with simple lightweight data structures, without locking. By comparison, other popular Web servers require some combination of *mmap*'ed files, spin-locks, IPC synchronization, and database connection pooling to achieve similar results.

At present, OKWS requires Web developers to program in C++, using the same SFS event library that undergirds all OKWS helper processes and core libraries. To simplify memory management, OKWS exposes SFS's reference-counted garbage collection scheme and high-level string library to the Web programmer. OKWS also provides a C++ preprocessor that allows for Perl-style "heredocs" and simplified template inclusion. Figure 3 demonstrates these facilities.

5.2 Asynchronous Database Proxies

OKWS provides Web developers with a generic library for translating between asynchronous RPC and any given blocking client library, in a manner similar to Flash's helper processes [23], and "manual calling automatic" in [1]. OKWS users can thus simply implement *database proxies*: asynchronous RPC front-ends to standard databases, such as MySQL [21] or Berkeley DB [29]. Our libraries provide the illusion of a standard asynchronous RPC dispatch routine. Internally, these proxies are multi-threaded and can block; the library handles synchronization and scheduling.

Database proxies employ a small and static number of worker threads and do not expand their thread pool. The intent here is simply to overlap requests to the underlying data source so that it might overlap its disk accesses and

```
void my_srvc_t::process ()
{
  str color = param["color"];
  /*o
    print (resp) <<EOF;
<html>
 <head>
  <title>${param["title"]}</title>
 </head>
EOF
    include (resp, "/body.html",
             { COLOR => ${color}});
  o*/
  output (resp);
}
```

Figure 3: Fragment of a Web service programmed in OKWS. The remote client supplies the title and color of the page via standard CGI-style parameter passing. The runtime templating system substitutes the user's choice of color for the token `COLOR` in the template `/body.html`. The variable `my_svc_t::resp` represents a buffer that collects the body of the HTTP response and then is flushed to the client via `output()`. With the `FilterCGI` flag set, OKWS filters all dangerous metacharacters from the `param` associative array.

benefit from disk arm scheduling.

Database proxies ought to run on the database machines themselves. Such a configuration allows the site administrator to "lock down" a socket-based database server, so that only local processes can execute arbitrary database commands. All other machines in the cluster—such as the Web server machines—only see the structured, and thus restricted, RPC interface exposed by the database proxy.

Finally, database proxies employ a simple mechanism for authenticating Web services. After a Web service connects to a database proxy, it supplies a 20-byte authentication token in a login message. The database proxy then grants the Web service permission to access a set of RPCs based on the supplied authentication token.

To facilitate development of OKWS database proxies, we wrapped MySQL's standard C library in an interface more suitable for use with SFS's libraries. We model our MySQL interface after the popular Perl DBI interface [24] and likewise transparently support both parsed and prepared SQL styles. Figure 4 shows a simple database proxy built with this library.

5.3 Real-World Experience

The author and two other programmers built a commercial Web site using the OKWS system in six months [22]. We were assisted by two designers who knew little C++ but made effective use of the HTML templating system. The application is Internet dating, and the site features a typical suite of services, including local matching, global matching, messaging, profile maintenance, site statistics, and picture browsing. Almost a million users have established accounts on the site, and at peak times, thousands of users maintain active sessions. Our current implementation uses 34 Web services and 12 database proxies.

We have found the system to be usable, stable and well-performing. In the absence of database bottlenecks or latency from serving advertisements, OKWS feels very responsive to the end user. Even those pages that require iterative computations—like match computations—load instantaneously.

Our Web cluster currently consists of four load balanced OKWS Web server machines, two read-only cache servers, and two read-write database servers, all with dual Pentium 4 processors. We use multiple OKWS machines only for redundancy; one machine can handle peak loads (about 200 requests per second) at about 7% CPU utilization, even as it *gzip*s most responses. A previous incarnation of this Web site required six ModPerl/Apache servers [20] to accommodate less traffic. It ultimately was abandoned due to insufficient software tools and prohibitive hardware and hosting expenses [30].

5.4 Separating Static From Dynamic

OKWS relies on other machines running standard Web servers to distribute static content. This means that all pages generated by OKWS should have only absolute links to external static content (such as images and style sheets), and OKWS has no reason to support keep-alive connections [11]. The servers that host static content for OKWS, however, can enable HTTP keep-alive as usual.

We note that serving static and dynamic content from different machines is already a common technique for performance reasons; administrators choose different hardware and software configurations for the two types of workloads. Moreover, static content service does not require access to sensitive site data and can therefore happen outside of a firewalled cluster, or perhaps at a different hosting facility altogether. Indeed, some sites push static content out to external distribution networks such as Akamai [2].

In our commercial deployment, we host a cluster of OKWS and database machines at a local colocation facility; we require hands-on hardware access and a network configured for our application. We serve static content from leased, dedicated servers at a remote facility where bandwidth is significantly cheaper.

```
struct user_xdr_t {
  string name<30>;
  int age;
};

// can only occur at initialization time
q = mysql->prepare (
      "SELECT age,name FROM tab WHERE id=?");

id = 1; // get ID from client
user_xdr_t u;
stmt = q->execute (id); // might block!
stmt->fetch (&u.age, &u.name);
reply (u);
```

Figure 4: Example of database proxy code with MySQL wrapper library. In this case, the Web developer is loading SQL results directly into an RPC XDR structure.

6 Security Discussion

In this section we discuss OKWS's security benefits and shortcomings.

6.1 Security Benefits

(1) The Local Filesystem. An OKWS service has almost no access to the file system when execution reaches custom code. If compromised, a service has write access to its coredump directory and can read from OKWS shared libraries. Otherwise, it cannot access `setuid` executables, the binaries of other OKWS services, or core dumps left behind by crashed OKWS processes. It cannot overwrite HTTP logs or HTML templates. Other OKWS services such as *oklogd* and *pubd* have more privileges, enabling them to write to and read from the file system, respectively. However, as OKWS matures, these helpers should not present security risks since they do not run site-specific code.

(2) Other Operating System Privileges. Because OKWS runs logically separate processes under different user IDs, compromised processes (with the exception of *okld*) do not have the ability to *kill* or *ptrace* other running processes. Similarly, no process save for *okld* can bind to privileged ports.

(3) Database Access. As described, all database access in OKWS is achieved through RPC channels, using independent authentication mechanisms. As a result, an attacker who gains control of an OKWS web service can only interact with the database in a manner specified by the RPC protocol declaration; he does not have generic SQL client access. Note that this is a stronger restriction than simple database permission systems alone can guarantee. For instance, on PHP systems, a particular service might only have SELECT permissions to a database's USERS table. But with control of the PHP server, an attacker could still issue commands like SELECT * FROM USERS. With OKWS, if the RPC protocol restricts access to row-wise queries and the keyspace of the table is sparse, the attacker has significantly more difficulty "mining" the database.[1]

OKWS's separation of code and privileges further limits attacks. If a particular service is compromised, it can establish a new connection to a remote RPC database proxy; however, because the service has no access to source code, binaries, or *ptraces* of other services, it knows no authentication tokens aside from its own.

Finally, OKWS database libraries provide runtime checks to ensure that SQL queries can be prepared only when a proxy starts up, and that all parameters passed to queries are appropriately escaped. This check insulates sloppy programmers from the "SQL injection" attacks mentioned in Section 2.2. We expect future versions of OKWS to enforce the same invariants at compile time.

(4) Process Isolation and Privilege Separation. OKWS is careful to separate the traditionally "buggy" aspects of Web servers from the most sensitive areas of the system. In particular, those processes that do the majority of HTTP parsing (the OKWS services) have the fewest privileges. By the same logic, *okld*, which runs as superuser, does no message parsing; it responds only to signals. For the other helper processes, we believe the RPC communication channels to be less error-prone than standard HTTP messaging and unlikely to allow intruders to traverse process boundaries.

Process isolation also limits the scope of those DoS attacks that exploit bugs in site-specific logic. Since the operating system sets per-process limits on resources such as file descriptors and memory, DoS vulnerabilities should not spread across process boundaries. We could make stronger DoS guarantees by adapting "defensive programming" techniques [26]. Qie *et al.* suggest compiling rate-control mechanisms into network services, for dynamic prevention of DoS attacks. Their system is applicable within OKWS's architecture, which relegates each service to a single address space. The same cannot be said for those systems that spread equivalent functionality across multiple address spaces.

6.2 Security Shortcomings

The current implementation of OKWS supports only C++ for service development. OKWS programmers

```
<html><head><title>Test Result</title></head>
<body>
<?
  $db = mysql_pconnect("okdb.lcs.mit.edu");
  mysql_select_db("testdb", $db);
  $id = $HTTP_GET_VARS["id"];
  $qry = "SELECT x,y FROM tab WHERE x=$id";
  $result = mysql_query("$qry", $db);
  $myrow = mysql_fetch_row($result);
  print("QRY $id $myrow[0] $myrow[1]\n");
?>
</body>
</html>
```

Figure 5: PHP version of the null service

should use the provided "safe" strings classes when generating HTML output, and they should use only auto-generated RPC stubs for network communication; however, OKWS does not prohibit programmers from using unsafe programming techniques and can therefore be made more susceptible to buffer overruns and stack-smashing attacks. Future versions of OKWS might make these attacks less likely by supporting higher-level programming languages such as Python or Perl.

Another shortcoming of OKWS is that an adversary who compromises an OKWS service can gain access to in-memory state belonging to other users. Developers might protect against this attack by encrypting cache entries with a private key stored in an HTTP cookie on the client's machine. Encryption cannot protect against an adversary who can compromise and passively monitor a Web server.

Finally, independent aspects of the system might be vulnerable due to a common bug in the core libraries.

7 Performance Evaluation

In designing OKWS we decided to limit its process pool to a small and fixed size. In our evaluation, we tested the hypothesis that this decision has a positive impact on performance, examining OKWS's performance as a function of the number of active service processes. We also present and test the claim that OKWS can achieve high throughputs relative to other Web servers because of its smaller process pool and its specialization for dynamic content.

7.1 Testing Methodology

Performance testing on Web servers usually involves the SPECweb99 benchmark [31], but this benchmark is not well-suited for dynamic Web servers that disable Keep-Alive connections and redirect to other machines for static content. We therefore devised a simple benchmark that better models serving dynamic content in real-world deployments, which we call the *null service benchmark*. For each of the platforms tested, we implemented a *null service*, which takes an integer input from a client, makes a database SELECT on the basis of that input, and returns the result in a short HTML response (see Figure 5). Test clients make one request per connection: they connect to the server, supply a randomly chosen query, receive the server's response, and then disconnect.

7.2 Experimental Setup

All Web servers tested use a large database table filled with sequential integer keys and their 20-byte SHA-1 hashes [12]. We constrained our client to query only the first 1,000,000 rows of this table, so that the database could store the entire dataset in memory. Our database was MySQL version 4.0.16.

All experiments used four FreeBSD 4.8 machines. The Web server and database machines were uniprocessor 2.4GHz and 2.6GHz Pentium 4s respectively, each with 1GB of RAM. Our two client machines ran Dual 3.0GHz Pentium 4s with 2GB of RAM. All machines were connected via fast Ethernet, and there was no network congestion during our experiments. Ping times between the clients and the Web server measured around 250 μs, and ping times between the Web server and database machine measured about 150 μs.

We implemented our test client using the OKWS libraries and the SFS toolkit. There was no resource strain on the client machines during our tests.

7.3 OKWS Process Pool Tests

We experimentally validated OKWS's frugal process allocation strategy by showing that the alternative—running many processes per service—performs worse. We thus configured OKWS to run a single service as a variable number of processes, and collected throughput measurements (in requests per second) over the different configurations. The test client was configured to simulate either 500, 1,000 or 2,000 concurrent remote clients in the different runs of the experiment.

Figure 6 summarizes the results of this experiment as the number of processes varied between 1 and 450. We attribute the general decline in performance to increased context-switching, as shown in Figure 7. In the single-process configuration, the operating system must switch between the null service and *okd*, the demultiplexing daemon. In this configuration, higher client concurrency implies fewer switches, since both *okd* and the null service have more outstanding requests to service before calling

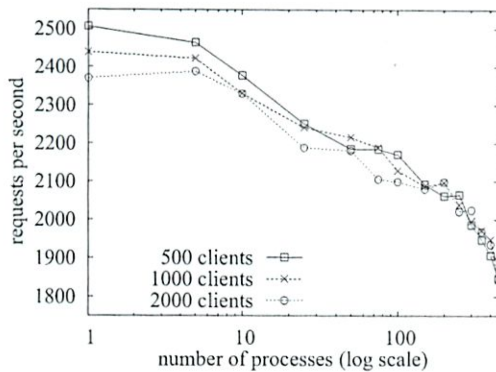


Figure 6: Throughputs achieved in the process pool test

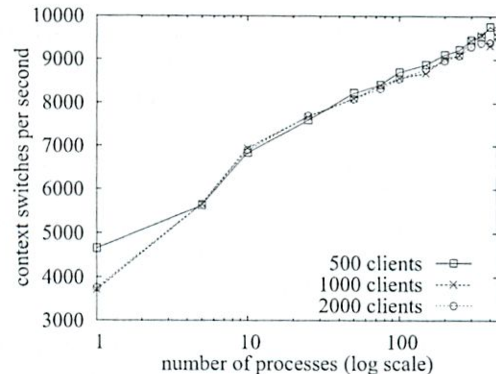


Figure 7: Context switching in the process pool test

sleep. This effect quickly disappears as the server distributes requests over more processes. As their numbers grow, each process has, on average, fewer requests to service per unit of time, and therefore calls *sleep* sooner within its CPU slice.

The process pool test supports our hypothesis that a Web server will consume more *computational* resources as its process pool grows. Although the experiments completed without putting *memory* pressure on the operating system, memory is more scarce in real deployments. The null service requires about 1.5MB of core memory, but our experience shows real OKWS service processes have memory footprints of at least 4MB, and hence we expect memory to limit server pool size. Moreover, in real deployments there is less memory to waste on code text, since in-memory caches on the Web services are crucial to good site performance and should be allowed to grow as big as possible.

7.4 Web Server Comparison

The other Web servers mentioned in Section 3.3—Haboob, Flash and Apache—are primarily intended for serving static Web pages. Because we have designed and tuned OKWS for an entirely dynamic workload, we hypothesize that when servicing such workloads, it performs better than its more general-purpose peers. Our experiments in this section test this hypothesis.

Haboob is Java-based, and we compiled and ran it with FreeBSD's native JDK, version 1.3. We tested Flash v0.1a, built with `FD_SETSIZE` set high so that Flash reported an ability to service 5116 simultaneous connections. Also tested was Apache version 2.0.47 compiled with multi-threading support and running PHP version 4.3.3 as a dynamic shared object. We configured Apache to handle up to 2000 concurrent connections. We ran OKWS in its standard configuration, with a one-to-one

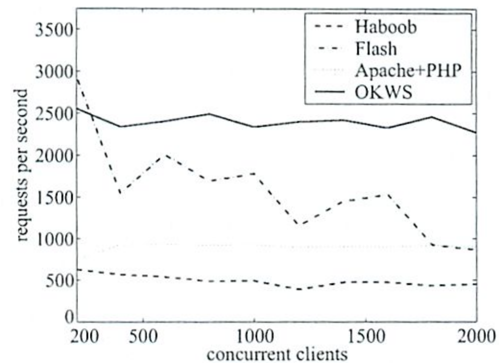


Figure 8: Throughputs for the single-service test

correspondence between processes and services.

We enabled HTTP access logging on all systems with the exception of Haboob, which does not support it. All systems used persistent database connections.

7.4.1 Single-Service Workload

In the single-service workload, clients with negligible latency request a dynamically generated response from the null service. This test entails the minimal number of service processes for OKWS and Flash and therefore should allow them to exhibit maximal throughput. By contrast, Apache and Haboob's process pools do not vary in size with the number of available services. We examined the throughput (Figure 8) and responsiveness (Figure 9) of the four systems as client concurrency increased. Figure 10 shows the cumulative distribution of client latencies when 1,600 were active concurrently.

Of the four Web servers tested, Haboob spent the most CPU time in user mode and performed the slowest. A likely cause is the sluggishness of Java 1.3's memory management.

When servicing a small number of concurrent clients, the Flash system outperforms the others; however, its per-

formance does not scale well. We attribute this degradation to Flash's CGI model: because custom-written Flash helper processes have only one thread of control, each instantiation of a helper process can handle only one external client. Thus, Flash requires a separate helper process for each external client served. At high concurrency levels, we noted a large number of running processes (on the order of 2000) and general resource starvation. Flash also puts additional strain on the database, demanding one active connection per helper—thousands in total. A database pooling system might mitigate this negative performance impact. Flash's results were noisy in general, and we can best explain the observed non-monotonicity as inconsistent operating system (and database) behavior under heavy strain.

Apache achieves 37% of OKWS's throughput on average. Its process pool is bigger and hence requires more frequent context switching. When servicing 1,000 concurrent clients, Apache runs around 450 processes, and context switches about 7500 times a second. We suspect that Apache starts queuing requests unfairly above 1,000 concurrent connections, as suggested by the plateau in Figure 9 and the long tail in Figure 10.

In our configuration, PHP makes frequent calls to the *sigprocmask* system call to serialize database accesses among kernel threads within a process. In addition, Apache makes frequent (and unnecessary) file system accesses, which though serviced from the buffer cache still entail system call overhead. OKWS can achieve faster performance because of a smaller process pool and fewer system calls.

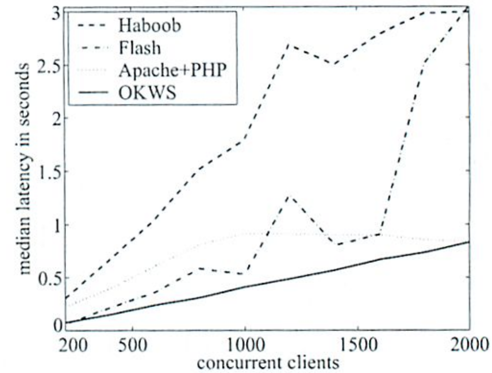


Figure 9: Median Latencies

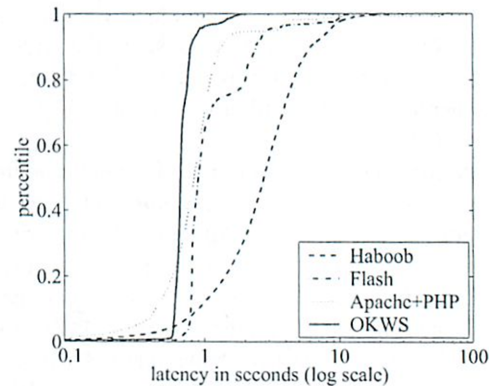


Figure 10: Client latencies for 1,600 concurrent clients

7.4.2 Many-Service Workload

In attempt to model a more realistic workload, we investigated Web servers running more services, serving more data, as experienced by clients over the WAN. We modified our null services to send out an additional 3000 bytes of text with every reply (larger responses would have saturated the Web server's access link in some cases). We made 10 uniquely-named copies of the new null service, convincing the Web servers that they were serving 10 distinct services. Finally, our clients were modified to pause an average of 75 ms between establishing a connection and sending an HTTP request. We ran the experiment from 200 to 2000 simultaneous clients, and observed a graph similar in shape to Figure 8.

Achieved throughputs are shown in Table 2 and are compared to the results observed in the single-service workload. Haboob's performance degrades most notably, probably because the many-service workload demands more memory allocations. Flash's throughput decreases by 23%. We observed that for this workload, Flash requires even more service processes, and at times over 2,500 were running. When we switched from the single-service to the many-service configuration, the number of OKWS service processes increased from 1 to 10. The results from Figure 6 show this change has little impact on throughput. We can better explain OKWS's diminished performance by arguing that larger HTTP responses result in more data shuffling in user mode and more pressure on the networking stack in kernel mode. The same explanation applies for Apache, which experienced a similar performance degradation.

8 Related work

Apache's [3] many configuration options and modules allow Web programmers to extend its functionality with a variety of different programming languages. However, neither 1.3.x's multi-process architecture nor 2.0.x's multi-threaded architecture is conducive to process isolation. Also, its extensibility and mushrooming code base make its security properties difficult to reason about.

	Haboob	Apache	Flash	OKWS
1 Service	490	895	1,590	2,401
10 Services	225	760	1,232	2,089
Change	−54.0%	−15.1%	−22.5%	−13.0%

Table 2: Average throughputs in connections per second

Highly-optimized event-based Web servers such as Flash [23] and Zeus [47] have eclipsed Apache in terms of performance. While Flash in particular has a history of outstanding performance serving static content, our performance studies here indicate that its architecture is less suitable for dynamic content. In terms of process isolation, one could most likely implement a similar separation of privileges in Flash as we have done with OKWS.

FastCGI [10] is a standard for implementing long-lived CGI-like helper processes. It allows separation of functionality along process boundaries but neither articulates a specific security policy nor specifies the mechanics for maintaining process isolation in the face of partial server compromise. Also, FastCGI requires the leader process to relay messages between the Web service and the remote client. OKWS passes file descriptors to avoid the overhead associated with FastCGI's relay technique.

The Haboob server studied here is one of many possible applications built on SEDA, an architecture for event-based network servers. In particular, SEDA uses serial event queues to enforce fairness and graceful degradation under heavy load. Larger systems such as Ninja [33] build on SEDA's infrastructure to create clusters of Web servers with the same appealing properties.

Other work has used the SFS toolkit to build static Web Servers and Web proxies [46]. Though the current OKWS architecture is well-suited for SMP machines, the adoption of *libasync-mp* would allow for finer-grained sharing of a Web workload across many CPUs.

OKWS uses events but the same results are possible with an appropriate threads library. An expansive body of literature argues the merits of one scheme over the other, and most recently, Capriccio's authors [34] argue that threads can achieve the same performance as events in the context of Web servers, while providing programmers with a more intuitive interface. Other recent work suggests that threads and events can coexist [1]. Such techniques, if applied to OKWS, would simplify stack management for Web developers.

In addition to the PHP [25] scripting language investigated here, many other Web development environments are in widespread use. Zope [48], a Python-based platform, has gained popularity due to its modularity and support for remote collaboration. CSE [13] allows developers to write Web services in C++ and uses some of the same sandboxing schemes we use here to achieve fault isolation. In more commercial settings, Java-based systems often favor thin Web servers, pushing more critical tasks to application servers such as JBoss [15] and IBM WebSphere [14]. Such systems limit a Web server's access to underlying databases in much the same way as OKWS's database proxies. Most Java systems, however, package all aspects of a system in one address space with many threads; our model for isolation would not extend to such a setting. Furthermore, our experimental results indicate significant performance advantages of compiled C++ code over Java systems.

Other work has proposed changes to underlying operating systems to make Web servers fast and more secure. The Exokernel operating system [16] allows its Cheetah Web server to directly access the TCP/IP stack, in order to reduce buffer copies allow for more effective caching. The Denali isolation kernel [45] can isolate Web services by running them on separate virtual machines.

9 Summary and Future Work

OKWS is a toolkit for serving dynamic Web content, and its architecture fits naturally into a compelling security model. The system's separation of processes provides reasonable assurances that vulnerabilities in one aspect of the system do not metastasize. The performance results we have seen are encouraging: OKWS derives significant speedups from a small and fixed process pool, lightweight synchronization mechanisms, and avoidance of unnecessary system calls. In the future, we plan to experiment with high-level language support and better resilience to DoS attacks. Independent of future improvements, OKWS is stable and practical, and we have used it to develop a popular commercial product.

Acknowledgments

I am indebted to David Mazières for his help throughout the project, and to my advisor Frans Kaashoek for help in preparing this paper. Michael Walfish significantly improved this paper's writing and presentation. My shepherd Eddie Kohler suggested many important improvements, Robert Morris and Russ Cox assisted in debugging, and the anonymous reviewers provided insightful comments. I thank the programmers, designers and others at OkCupid.com—Patrick Crosby, Jason Yung, Chris Coyne, Christian Rudder and Sam Yagan—for adopting and improving OKWS, and Jeremy Stribling and Sarah Friedberg for proofreading. This research was

supported in part by the DARPA Composable High Assurance Trusted Systems program (BAA #01-24), under contract #N66001-01-1-8927.

Availability

OKWS is available under an open source license at www.okws.org.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.

[2] Akamai Technologies, Inc. http://www.akamai.com.

[3] The Apache Software Foundation. http://www.apache.org.

[4] Apache Tutorial: Introduction to Server Side Includes. http://httpd.apache.org/docs/howto/ssi.html.

[5] Bugtraq ID 4606. SecurityFocus. http://www.securityfocus.com/bid/4606/info/.

[6] Bugtraq ID 5993. SecurityFocus. http://www.securityfocus.com/bid/5993/info/.

[7] Bugtraq ID 7255. SecurityFocus. http://www.securityfocus.com/bid/7255/info/.

[8] Bugtraq ID 8138. SecurityFocus. http://www.securityfocus.com/bid/8138/info/.

[9] CERT® Coordination Center. http://www.cert.org.

[10] Open Market. Fastcgi. http://www.fastcgi.com.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1*. Internet Network Working Group RFC 2616, 1999.

[12] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfi eld, VA, April 1995.

[13] T. Gchwind and B. A. Schmit. CSE —a C++ servlet environment for high-performance web applications. In *Proceedings of the FREENIX Track: 2003 USENIX Technical Conference*, San Antonio, TX, 2003. USENIX.

[14] IBM corporation. IBM websphere application server. http://www.ibm.com.

[15] JBoss Group. http://www.jboss.org.

[16] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM.

[17] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.

[18] D. Mazières. A toolkit for user-level fi le systems. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.

[19] Microsoft Corporation. IIS. http://www.microsoft.com/windowsserver2003/iis/default.mspx.

[20] mod_perl. http://perl.apache.org.

[21] MySQL. http://www.mysql.com.

[22] OkCupid.com. http://www.okcupid.com.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An effi cient and portable Web server. In *Proceedings of the 1999 USENIX*, Monterey, CA, June 1999. USENIX.

[24] Perl DBI. http://dbi.perl.org.

[25] PHP: Hypertext processor. http://www.php.net.

[26] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.

[27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, 1975.

[28] SecurityFocus. http://www.securityfocus.com.

[29] Sleepycat Software. http://www.sleepycat.com.

[30] The SparkMatch service. Previously available at http://www.thespark.com.

[31] Standard performance evaluation corporation. the specweb99 benchmark. http://www.spec99.org/osg/web99/.

[32] R. Srinivasan. RPC: Remote procedure call protocol specifi cation version 2. RFC 1831, Network Working Group, August 1995.

[33] J. R. van Berhen, E. A. Brewer, N. Borisova, M. C. an Matt Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.

[34] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.

[35] Vulnerability CAN-2001-1246. SecurityFocus. http://www.securityfocus.com/bid/2954/info/.

[36] Vulnerability CAN-2002-0656. SecurityFocus. http://www.securityfocus.com/bid/5363/info/.

[37] Vulnerability CAN-2003-0132. SecurityFocus. http://www.securityfocus.com/bid/7254/info/.

[38] Vulnerability CAN-2003-0253. http://www.securityfocus.com/bid/8137/info/.

[39] Vulnerability CAN-2003-0542. SecurityFocus. http://www.securityfocus.com/bid/8911/info/.

[40] Vulnerability CVE-2002-0061. SecurityFocus. http://www.securityfocus.com/bid/4435/info/.

[41] Vulnerability Note VU117243. CERT. http://www.kb.cert.org/vuls/id/910713.

[42] Vulnerability Note VU91073. CERT. http://www.kb.cert.org/vuls/id/910713.

[43] Vulnerability Note VU979793. CERT. http://www.kb.cert.org/vuls/id/979793.

[44] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[45] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.

[46] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX*, San Antonio, TX, June 2003. USENIX.

[47] Zeus Technology Limited. Zeus Web Server. http://www.zeus.co.uk.

[48] The Zope Corporation. http://www.zope.org.

Notes

[1] Similar security properties are possible with a standard Web server and a database that supports stored procedures, views, and roles.

6.858: Computer Systems Security

Fall **2012**

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 3

What's the worst that could happen if one service in OKWS were to leak its 20-byte database proxy authentication token?

(it would have all privileges that service had

if that included delete - could wipe

(that's basic - but these seem easy...

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Monday, 10-Sep-2012 18:58:14 EDT*

8/12

Paper Question 3

Michael Plasmeier

The attacker would have all privileges that the service had. If that includes delete privileges, the attacker could wipe out the database.

L3

Web Server Security

9/12

How to design better secured web server
OTWF

The lab web sever is similar to this

how to limit damage of compromised system

---

building blocks: Unix Protection Mechanisms

Unix not designed for security
So not flexible, elegant

think about

- Principals: - actors
- Objects:
- Operations:
- Who decides:

(2)

Principals

User ID

32 bit

Group ID

32 bit

Processes

Are the actors

Has one UID

A set of 0 or more GIDs

Special Super User

UID = 0

"root"

no security checks - do anything

3) Objects

Files, directories ← inode
↳ lookup, list, create, unlink
↳ read, write, execute, change permission, link

Each inode has a set of permissions

Owner uid → r, w, x ← execute
Owner gid → r, w, x
Other → r, w, x

Octal base #s

r	w	x
4	2	1

To change permission
must be same uid
gid does not matter

To create hard link
to write to it
(Prof: ~~[illegible]~~ ~~[illegible]~~ there is no good reason why)

(4)

Some trick about execute on directory

~~missed / not important~~

So open("/etc/passwd")

↑ need execute on directory to look it up

↑ need read permission

This is not super general

How do restriction to ~~[illegible]~~ intersection of 2 groups

Create /a/b/c.txt

↑ group 1 only readable

↑ group 2 only readable

So must be able to read a <u>and</u> b

<u>Other objects</u>

File descriptor

Open() → 5

↑ file descriptor for that file

(5)

Once have file id can do anything
Security checks only on open

Allows fine grained FD passing
T(i)

Processes

Kill, signal, debug (ptrace)
are uids on processes the same?

Memory

Not shared generally
Nowdays some exceptions
- ptrace
  - same uid
- memory-mapped file
  - checks file permissions

⑥

Networking

Was added on later

So don't really use same permission mechanisms

bind/accept

< 1024 port

Only if UID = 0

So only admins

So have to be root for new users

or setting up low ports

So many apps need root

If 1 is compromised, has full access

Also send/rec raw packets requires root

7

Firewall

Does not ~~[illegible]~~ look at Uid

Windows is per-process

Not in Linux

How set all these permissions up front?

Set uid()
Set gid()
Set groups()

) Can only do if Uid = 0

But if want to have a lot of uids
Need components running as root
Which is why many apps don't do this
paper Mon trys to fix

Login serv
Starts as Uid = 0
Set uid
runs shell

How do you regain privileges

Say want to start web server - which needs root

So setuid binaries

SU = switch user ids

└ at /usr/bin/su

Sets special flag in inode bit

if 0 is owner of binary

Owner = you, whoever created file

readers = some groups

extra bit up front to run as that user id

[4] 644

Somewhat problematic

└ can't be any buffer overflows

└ so must trust it

9

Shared libs had some vunerabilites
(missed term) — fairly trusting

Was a big issue originally

how prevent it from being exected?
↙ Can't just use user id

Chroot

effectivly changes file system namespace

chroot("/var/okws/run")

Then if try open("/etc/passwd")

it will run open("/var/okws/run/etc/passwd")

So put only the files it needs inside there?

if had link → will still get through

but static links would get stopped

(10)

How to build a web server?

Apache

typical design
not particularly secure

runs as one user id
∟ like www-data

first starts as root though to get pt 80
then calls set user id
retains access to file descriptor for pt 80

Often the apps PHP, Python also share
the same uid

Can leak sensitive files
∟ any files in apache's path
Buffer overflows
Application level bugs ← esp since not well tested

(11)

SQL Injection

4/8/?

DB has own security plan

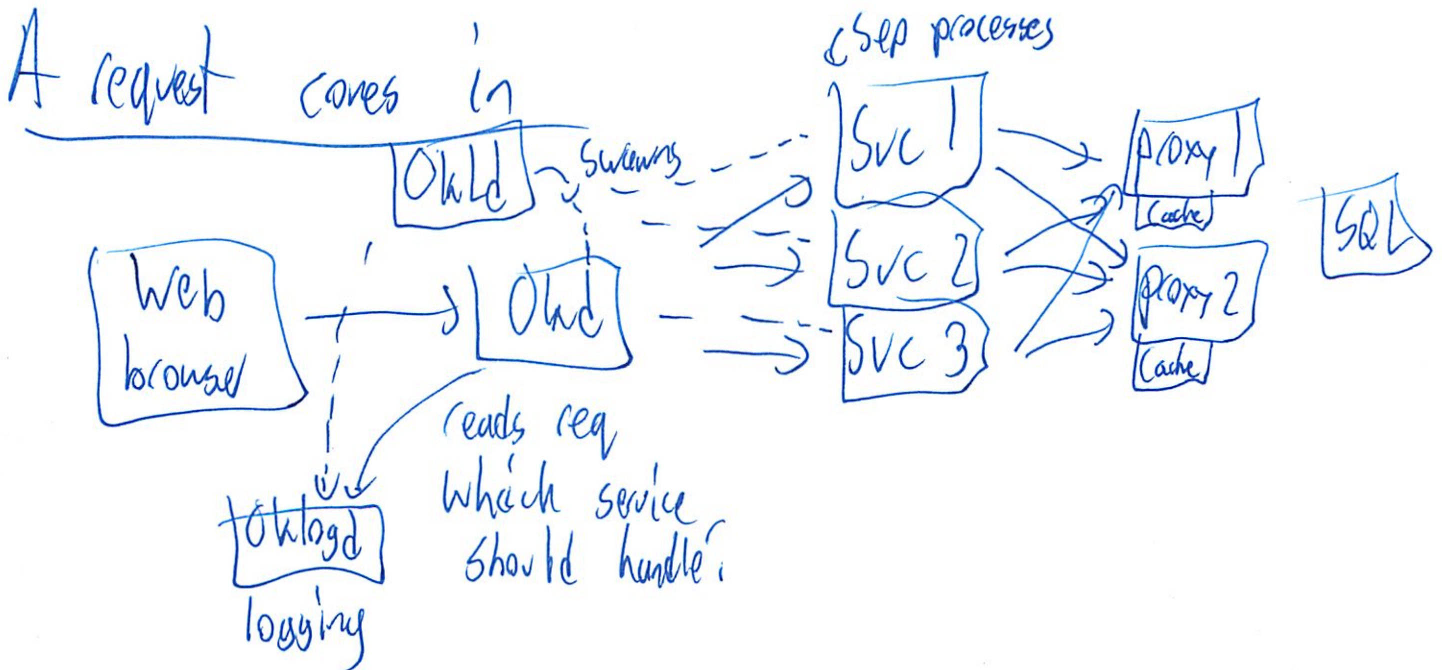
But we have 1 user that has all access

OkWS

From Ok Cupid

Wanted to hire less-skilled programmers

Lots of hoops to jump through for OkWS



(12)

proxy limit access to db

Capture db queries

So like enforced SQL templates

q1 SELECT * FROM x ~~where~~ WHERE id = (*)

Write proxy to santize input

So if leak 20 byte token

Can issue all SQL queries

Want to firewall off backend machine

So also need to ~~com~~ compromise front-end

How isolate everything?

OKWS spawns everything and sets uids correctly

└ runs as root all the time

because it needs to set uids

and bind to port 80

(13)

also wanted log file to be append only
 └ Unix does not support

 So okld implements this

also chroot those componts to diff directories
 the services share it
 - for efficency
 - since need a copy of all files
 - just the app code you can prob download

okld
pubd) are seperate

 - not particularly principled

Why does okld stay as root?
 In case the services crash
 So can ~~crash~~ restart them

If a service is compromised, it should be floated

(14)

So want it to run, not writable

So

/var/lotus/run/svc/

Owner uid = 0 ← only read [400]

Owner gid = 51000 ← only execute

So permissions can't be modified

~~if each service had root access~~
~~could give ex~~

(confused)

Prot it harder to accidently screw up

Could change permission

But must do extra work to screw up

(15)

Goal: limit damage

Is it better than Apache?
Static files -> not really
app code ->

Performance is not a problem

OKld
?If compromised

OKld should not take input
What is the attack surface?
Just waits for children to exit, then restarts it
Could perhaps race condition that corrupts memory

Or denial of service from crashing
Could also be crashing since trying to guess
stack canary

(16)

Okd]

: Perhaps send commands to wrong service

Could keep running + monitor traffic

Could see user password

One string parse ← attack surface

Oklogd]

Could lose log

But attack surface is fairly small

Storing cookie - even if encrypted

~~[illegible]~~ can reuse it

Svcs]

will be able to run templitized SQL query

have to parse full HTTP req ← attack surface

kernel)

Enforcement provided by linux kernel

hope no buffer overflows!

But can root your android

(17)

Only OkCupid uses it
└ but still uses it

~~[illegible]~~

Still SQL injections

If in web framework

Most part is coming up w/ proxy
SQL templates

L3 9/12

```
OKWS
====

Administrivia: part 1 of lab 1 due this Friday.

Today's lecture: how to build a secure web server on Unix.
  The design of our lab web server, zookws, is inspired by OKWS.

Background: security and protection in Unix
  Typical principals: user IDs, group IDs (32-bit integers).
    Each process has a user ID (uid), and a list of group IDs (gid + grouplist).
    For mostly-historical reasons, a process has a gid + extra grouplist.
    Superuser principal (root) represented by uid=0, bypasses most checks.
  What are the objects + ops in Unix, and how does the OS do access control?
    Files, directories.
      File operations: read, write, execute, change perms, ..
      Directory operations: lookup, create, remove, rename, change perms, ..
      Each inode has an owner user and group.
      Each inode has read, write, execute perms for user, group, others.
      Typically represented as a bit vector written base 8 (octal);
        octal works well because each digit is 3 bits (read, write, exec).
      Who can change permissions on files?  Only user owner (process UID).
      Hard link to file: need write permission to file (perhaps due to quotas).
      Execute for directory means being able to lookup names (but not ls).
      Checks for process opening file /etc/passwd:
        Must be able to look up 'etc' in /, 'passwd' in /etc.
        Must be able to open /etc/passwd (read or read-write).
      Suppose you want file readable to intersection of group1 and group2.
        Is it possible to implement this in Unix?
    File descriptors.
      File access control checks performed at file open.
      Once process has an open file descriptor, can continue accessing.
      Processes can pass file descriptors (via Unix domain sockets).
    Processes.
      What can you do to a process?
        debug (ptrace), send signal, wait for exit & get status, ..
      Debugging, sending signals: must have same UID (almost).
        Various exceptions, this gets tricky in practice.
      Waiting / getting exit status: must be parent of that process.
    Memory.
      One process cannot generally name memory in another process.
      Exception: debug mechanisms.
      Exception: memory-mapped files.
    Networking.
      Operations:
        bind to a port
        connect to some address
        read/write a connection
        send/receive raw packets
      Rules:
        - only root (UID 0) can bind to ports below 1024;
          (e.g., arbitrary user cannot run a web server on port 80.)
        - only root can send/receive raw packets.
        - any process can connect to any address.
        - can only read/write data on connection that a process has an fd for.
      Additionally, firewall imposes its own checks, unrelated to processes.
  How does the principal of a process get set?
    System calls: setuid(), setgid(), setgroups().
    Only root (UID 0) can call these system calls (to first approximation).
  Where does the user ID, group ID list come from?
    On a typical Unix system, login program runs as root (UID 0)
    Checks supplied user password against /etc/shadow.
```

```
    Finds user's UID based on /etc/passwd.
    Finds user's groups based on /etc/group.
    Calls setuid(), setgid(), setgroups() before running user's shell
  How do you regain privileges after switching to a non-root user?
    Could use file descriptor passing (but have to write specialized code)
    Kernel mechanism: setuid/setgid binaries.
      When the binary is executed, set process UID or GID to binary owner.
      Specified with a special bit in the file's permissions.
      For example, su / sudo binaries are typically setuid root.
      Even if your shell is not root, can run "su otheruser"
      su process will check passwd, run shell as otheruser if OK.
      Many such programs on Unix, since root privileges often needed.
    Why might setuid-binaries be a bad idea, security-wise?
      Many ways for adversary (caller of binary) to manipulate process.
      In Unix, exec'ed process inherits environment vars, file descriptors, ..
      Libraries that a setuid program might use not sufficiently paranoid
      Historically, many vulnerabilities (e.g. pass $LD_PRELOAD, ..)
  How to prevent a malicious program from exploiting setuid-root binaries?
    Kernel mechanism: chroot
      Changes what '/' means when opening files by path name.
      Cannot name files (e.g. setuid binaries) outside chroot tree.
    For example, OKWS uses chroot to restrict programs to /var/okws/run, ..
    Kernel also ensures that '/../' does not allow escape from chroot.
    Why chroot only allowed for root?
      1. setuid binaries (like su) can get confused about what's /etc/passwd.
      2. many kernel implementations (inadvertently?) allow recursive calls
         to chroot() to escape from chroot jail, so chroot is not an effective
         security mechanism for a process running as root.
    Why hasn't chroot been fixed to confine a root process in that dir?
      Root can write kern mem, load kern modules, access disk sectors, ..

Background: traditional web server architecture (Apache).
  Apache runs N identical processes, handling HTTP requests.
  All processes run as user 'www'.
  Application code (e.g. PHP) typically runs inside each of N apache processes.
  Any accesses to OS state (files, processes, ...) performed by www's UID.
  Storage: SQL database, typically one connection with full access to DB.
    Database principal is the entire application.
  Problem: if any component is compromised, adversary gets all the data.
  What kind of attacks might occur in a web application?
    Unintended data disclosure (getting page source code, hidden files, ..)
    Remote code execution (e.g., buffer overflow in Apache)
    Buggy application code (hard to write secure PHP code), e.g. SQL inj.
    Attacks on web browsers (cross-site scripting attacks)

Back to OKWS: what's their application / motivation?
  Dating web site: worried about data secrecy.
  Not so worried about adversary breaking in and sending spam.
  Lots of server-side code execution: matching, profile updates, ...
  Must have sharing between users (e.g. matching) -- cannot just partition.
  Good summary of overall plan:
    "aspects most vulnerable to attack are least useful to attackers".

Why is this hard?
  Unix makes it tricky to reduce privileges (chroot, UIDs, ..)
  Applications need to share state in complicated ways.
  Unix and SQL databases don't have fine-grained sharing control mechanisms.

How does OKWS partition the web server?
  Figure 1 in paper.
  How does a request flow in this web server?
    okd -> oklogd
        -> pubd
```

```
        -> svc -> dbproxy
               -> oklogd
  How does this design map onto physical machines?
    Probably many front-end machines (okld, okd, pubd, oklogd, svc)
    Several DB machines (dbproxy, DB)

How do these components interact?
  okld sets up socketpairs (bidirectional pipes) for each service.
    One socketpair for control RPC requests (e.g., "get a new log socketpair").
    One socketpair for logging (okld has to get it from oklogd first via RPC).
    For HTTP services: one socketpair for forwarding HTTP connections.
    For okd: the server-side FDs for HTTP services' socketpairs (HTTP+RPC).
  okd listens on a separate socket for control requests (repub, relaunch).
    Seems to be port 11277 in Figure 1, but a Unix domain socket in OKWS code.
    For repub, okd talks to pubd to generate new templates,
      then sends generated templates to each service via RPC control channel.
  Services talk to DB proxy over TCP (connect by port number).

How does OKWS enforce isolation between components in Figure 1?
  Each service runs as a separate UID and GID.
  chroot used to confine each process to a separate directory (almost).
  Components communicate via pipes (or rather, Unix domain socket pairs).
  File descriptor passing used to pass around HTTP connections.
  What's the point of okld?
  Why isn't okld the same as okd?
  Why does okld need to run as root?  (Port 80, chroot/setuid.)
  What does it take for okld to launch a service?
    Create socket pairs
    Get new socket to oklogd
    fork, setuid/setgid, exec the service
    Pass control sockets to okd
  What's the point of oklogd?
  What's the point of pubd?
  Why do we need a database proxy?
    Ensure that each service cannot fetch other data, if it is compromised.
      DB proxy protocol defined by app developer, depending on what app requires.
      One likely-common kind of proxy is a templatized SQL query.
      Proxy enforces overall query structure (select, update),
        but allows client to fill in query parameters.
    Where does the 20-byte token come from?  Passed as arguments to service.
    Who checks the token?  DB proxy has list of tokens (& allowed queries?)
    Who generates token?  Not clear; manual by system administrator?
    What if token disclosed?  Compromised component could issue queries.
  Table 1: why are all services and okld in the same chroot?  Is it a problem?
    How would we decide?  What are the readable, writable files there?
    Readable: shared libraries containing service code.
    Writable: each service can write to its own /cores/<uid>.
    Where's the config file?  /etc/okws_config, kept in memory by okld.
    oklogd & pubd have separate chroots because they have important state:
      oklogd's chroot contains the log file, want to ensure it's not modified.
      pubd's chroot contains the templates, want to avoid disclosing them (?).
  Why does OKWS need a separate GID for every service?
    Need to execute binary, but file ownership allows chmod.
    Solution: binaries owned by root, service is group owner, mode 0410.
    Why 0410 (user read, group execute), and not 0510 (user read & exec)?
  Why not process per user?  Is per user strictly better?  user X service?
    Per-service isolation probably made sense for okcupid given their apps.
      (i.e., perhaps they need a lot of sharing between users anyway?)
    Per-user isolation requires allocating UIDs per user, complicating okld,
      and reducing performance (though may still be OK for some use cases).

Does OKWS achieve its goal?
  What attacks from the list of typical web attacks does OKWS solve, and how?
```

```
      Most things other than XSS are addressed.
      XSS sort-of addressed through using specialized template routines.
    What's the effect of each component being compromised, and "attack surface"?
      okld: root access to web server machine, but maybe not to DB.
        attack surface: small (no user input other than svc exit).
      okd: intercept/modify all user HTTP reqs/responses, steal passwords.
        attack surface: parsing the first line of HTTP request; control requests.
      pubd: corrupt templates, leverage to maybe exploit bug in some service?
        attack surface: requests to fetch templates from okd.
      oklogd: corrupt/ignore/remove/falsify log entries
        attack surface: log messages from okd, okld, svcs
      service: send garbage to user, access data for svc (modulo dbproxy)
        attack surface: HTTP requests from users (+ control msgs from okd)
      dbproxy: access/change all user data in the database it's talking to
        attack surface: requests from authorized services
                        requests from unauthorized services (easy to drop)
    OS kernel is part of the attack surface once a single service is compromised.
      Linux kernel vulnerabilities rare, but still show up several times a year.
    OKWS assumes developer does the right thing at design level (maybe not impl):
      Split web application into separate services (not clump all into one).
      Define precise protocols for DB proxy (otherwise any service gets any data).
    Performance?
      Seems better than most alternatives.
      Better performance under load (so, resists DoS attacks to some extent)
    How does OKWS compare to Apache?
      Overall, better design.
      okld runs as root, vs. nothing in Apache, but probably minor.
      Neither has a great solution to client-side vulnerabilities (XSS, ..)
    How might an adversary try to compromise a system like OKWS?
      Exploit buffer overflows or other vulnerabilities in C++ code.
      Find a SQL injection attack in some dbproxy.
      Find logic bugs in service code.
      Find cross-site scripting vulnerabilities.

  How successful is OKWS?
    Problems described in the paper are still pretty common.
    okcupid.com still runs OKWS, but doesn't seem to be used by other sites.
    C++ might not be a great choice for writing web applications.
      For many web applications, getting C++ performance might not be critical.
      Design should be applicable to other languages too (Python, etc).
      Infact, zookws for labs in 6.858 is inspired by OKWS, runs Python code.
    DB proxy idea hasn't taken off, for typical web applications.
      But DB proxy is critical to restrict what data a service can access in OKWS.
      Why?  Requires developers to define these APIs: extra work, gets in the way.
      Can be hard to precisely define the allowed DB queries ahead of time.
        (Although if it's hard, might be a flag that security policy is fuzzy.)
    Some work on privilege separation for Apache (though still hard to use).
      Unix makes it hard for non-root users to manipulate user IDs.
      Performance is a concern (running a separate process for each request).
    scripts.mit.edu has a similar design, running scripts under different UIDs.
      Mostly worried about isolating users from one another.
      Paranoid web app developer can create separate locker for each component.
    Sensitive systems do partitioning at a coarser granularity.
      Credit card processing companies split credit card data vs. everything else.
      Use virtual machines or physical machine isolation to split apps, DBs, ..

  How could you integrate modern Web application frameworks with OKWS?
    Need to help okd figure out how to route requests to services.
    Need to implement DB proxies, or some variant thereof, to protect data.
      Depends on how amenable the app code is to static analysis.
      Or need to ask programmer to annotate services w/ queries they can run.
    Need to ensure app code can run in separate processes (probably OK).
```

References:
  http://pdos.csail.mit.edu/6.858/2012/readings/setuid.pdf
  http://httpd.apache.org/docs/trunk/suexec.html
  http://privsep.org/