Biologically Inspired Software Defenses

Michael Franz
Department of Computer Science
University of California, Irvine
Software Security

- “safe” programming languages have been around for decades
  - static type safety
  - mandatory array bounds checks
  - mandatory pointer null (NIL) checks
  - garbage collection

- unfortunately, decades later these lessons still haven’t been learned

- C/C++ still the backbone of systems programming

- this creates job security for computer security researchers
Problem: Critical Software Bugs

- many software vulnerabilities are “critical”
  - allow the instantaneous complete takeover of computers that have the bug

- bugs of this severity are discovered regularly in widely used software

- deeper question: are there “insiders” who know about the bugs?
Isn’t This Trivial & Old?

- one would think so, but unfortunately...
- buffer overflows and related pointer vulnerabilities still account for extremely serious exploits
- sometimes, vulnerabilities exist for extended periods
- e.g., 19 August 2009
  - Researchers uncovered a vulnerability in the Linux kernel that puts most versions built in the past eight years at risk of complete takeover. Affects all 2.4 and 2.6 kernels since 2001 on all architectures.

  The bug involves the way kernel-level routines such as sock_sendpage react when they are left unimplemented. Instead of linking to a corresponding placeholder, (for example, sock_no_accept), the function pointer is left uninitialized. Sock_sendpage doesn’t always validate the pointer before dereferencing it, leaving the OS open to local privilege escalation that can completely compromise the underlying machine.
Wide-Range Vulnerabilities

Microsoft Security Bulletin MS13-002 - Critical
Vulnerabilities in Microsoft XML Core Services Could Allow Remote Code Execution (2756145)
Published: Tuesday, January 08, 2013 | Updated: Tuesday, January 08, 2013
Version: 1.1

General Information

Executive Summary
This security update resolves two privately reported vulnerabilities in Microsoft XML Core Services. The vulnerabilities could allow remote code execution if a user views a specially crafted webpage using Internet Explorer. An attacker would have no way to force users to visit such a website. Instead, an attacker would have to convince users to visit the website, typically by getting them to click a link in an email message or Instant Messenger message that takes the user to the attacker's website.

Security Advisory for Adobe Flash Player, Adobe Reader and Acrobat

Release date: April 11, 2011

Vulnerability identifier: APSA11-02

CVE number: CVE-2011-0611

Platform: See "Affected software versions" section below for details

Summary

A critical vulnerability exists in Flash Player 10.2.153.1 and earlier versions (Adobe Flash Player 10.2.154.25 and earlier for Chrome users) for Windows, Macintosh, Linux and Solaris, Adobe Flash Player 10.2.156.12 and earlier versions for Android, and the Authplay.dll component that ships with Adobe Reader and Acrobat X (10.0.2) and earlier 10.x and 9.x versions for Windows and Macintosh operating systems.

This vulnerability (CVE-2011-0611) could cause a crash and potentially allow an attacker to take control of the affected system. There are reports that this vulnerability is being exploited in the wild in targeted attacks via a Flash (.swf) file embedded in a Microsoft Word (.doc) file delivered as an email attachment, targeting the Windows platform. At this time, Adobe is not aware of any attacks via PDF targeting Adobe Reader and Acrobat. Adobe Reader X Protected Mode mitigations would prevent an exploit of this kind from executing.
Summary

Cisco Unified IP Phones 7900 Series versions 9.3(1)SR1 and prior contain an arbitrary code execution vulnerability that could allow a local attacker to execute code or modify arbitrary memory with elevated privileges.

This vulnerability is due to a failure to properly validate input passed to kernel system calls from applications running in userspace. An attacker could exploit this issue by gaining local access to the device using physical access or authenticated access using SSH and executing an attacker-controlled binary that is designed to exploit the issue. Such an attack would originate from an unprivileged context.

Ang Cui initially reported the issue to the Cisco Product Security Incident Response Team (PSIRT). On November 6, 2012, the Cisco PSIRT disclosed this issue in Cisco bug ID CSCuc83860 (registered customers).
Vulnerability “Marketplace”

Abstract

Trading of 0-day computer exploits between hackers has been taking place for as long as computer exploits have existed. A black market for these exploits has developed around their illegal use. Recently, a trend has developed toward buying and selling these exploits as a source of legitimate income for security researchers. However, this emerging “0-day market” has some unique aspects that make this particularly difficult to accomplish in a fair manner. These problems, along with possible solutions will be discussed. These issues will be illustrated by following two case studies of attempted sales of 0-day exploits.

1 Introduction

There has long been a black market for computer exploits. For a long time, hackers were content to trade or sell exploits amongst themselves, mostly for prestige. Computer security researchers normally followed “responsible” disclosure which entails contacting the vendor and usually receiving acknowledgment when the vulnerability was announced along with the supplied patch. In the last few years, the market for 0-day exploits, those for which there is no available patch, has begun to migrate into the commercial space.

Certain companies sell tools or packages which contain zero day exploits [1, 2, 3], while some others purport to be able to broker deals between researchers and buyers [5]. Likewise, the illegal market for these exclusive tools has begun to become more economically based as spammers and criminals become interested in the use of 0-day exploits for use in illegal activity [24].

There is strong evidence that the best researchers are now motivated more by monetary gain than prestige [19, 20]. As a computer security researcher, there are many options available after discovering a vulnerability in a high-profile application or operating system. She may choose to report the vulnerability to the vendor, or simply announce it publicly without vendor notification. Such a choice may be made in order to increase her reputation or add to her resume. She may choose to sell the information on the black market, but faces potential criminal prosecution for such an action. Finally, she may choose to attempt to sell this information to a legitimate buyer. Such legal buyers may include government agencies, commercial tool suppliers, large penetration testing and consulting firms, intrusion detection companies, and subscription services. This paper documents the problems such a researcher will face when attempting to sell this vulnerability information or exploit to a legitimate buyer. It will then discuss possible solutions to some of these fundamental issues. Finally, these problems and solutions will be addressed in the context of two actual attempts at sell.
A “Market” In Unreported Bugs

Exploit Exchange

Exploit Exchange is the premier online marketplace for buying and selling security exploits. By providing a financial incentive to security researchers, high-quality security exploits are developed quickly and abundantly. The resulting market competition facilitates an efficient and transparent channel for software vendors, security researchers, penetration testers and ethical hackers alike to mitigate security vulnerabilities more effectively than current market conditions permit.

- trading knowledge in undisclosed software bugs is now a “legitimate” business
- the “bad guys” pay more than the “good guys”
- selling a vulnerability that you have found yourself to the highest bidder is perfectly legal in most jurisdictions
Each price assumes an exclusive sale, the most modern version of the software, and, of course, not alerting the software’s vendor. Some fees might even be paid in installments, with each subsequent payment depending on the vendor not patching the security vulnerabilities used by the exploit. In some cases the techniques would need to be used in combination to be effective.

An exploit’s price factors in both how widely the target software is used as well as the difficulty of cracking it. A technique that allows a hacker to gain control of a Mac OSX machine after hacking an application might earn only a fraction of one that targets Windows, for instance, because of Windows’ greater market share. But an iOS exploit pays more than one that targets Android devices partly because it requires defeating Apple’s significantly tougher security features. That means most agencies can simply develop their own Android attacks, the Grugq says, while ones that can penetrate the iPhone are rare and pricey. For the Jailbreakme 3 iOS exploit created by the hacker Comex last year, the Grugq says he heard agencies would have been eager to pay $250,000 for exclusive use of the attack.
Juergen Marester, a 24-year-old French network consultant, needed seed capital to start his own computer-security company. So he turned to his off-hours hobby -- black-hat hacking -- and did what a growing number of hackers are doing: selling "0days" (pronounced "oh-days" or "zero days," it generally refers to unknown, or zero-hour, software threats). These are recipes and code for penetrating the software run by governments, corporations, and private citizens. When properly deployed, 0days can result in minor disruptions such as a Web site’s temporary paralysis. At their extreme, they grant an attacker total control over a network.
On the tiny Mediterranean island of Malta, two Italian hackers have been searching for bugs — not the island's many beetle varieties, but secret flaws in computer code that governments pay hundreds of thousands of dollars to learn about and exploit.

The hackers, Luigi Auriemma, 32, and Donato Ferrante, 28, sell technical details of such vulnerabilities to countries that want to break into the computer systems of foreign adversaries. The two will not reveal the clients of their company, ReVuln, but big buyers of services like theirs include the National Security Agency — which seeks the flaws for America's growing arsenal of cyberweapons —
Is Open Source Better?

- no, the situation may be even more severe
- open source teams usually have limited resources and typically cannot afford to employ techniques such as high-end static analysis
- but an adversary with substantial resources could build a testing lab that systematically finds vulnerabilities — without anyone knowing
- hence, open-source software may lead to asymmetric threats in which an attacker knows more about a system than the system’s own creator
Adobe says source code, customer data stolen by hackers

By Joseph Menn

BOSTON | Thu Oct 3, 2013 8:41pm EDT

(Reuters) - Adobe Systems Inc said on Thursday that hackers had stolen source code to some of its most popular software and data about millions of its customers.

Security experts worry about the theft of source code because close review of the programs can lead to the discovery of new flaws that can be used to launch hard-to-detect attacks against all users of that software.

The hackers took source code for Adobe Acrobat, which is used to create electronic documents in the PDF format, as well as ColdFusion and ColdFusion Builder, used to create Internet applications, Adobe said.
Anatomy Of An Exploit

- successful exploit depends on two things:
  - the attacker knows about a place in which the program has an “unspecified” behavior
    - example: a particular buffer will actually accept input greater than its length
  - even though the behavior is “unspecified”, the attacker knows exactly what will happen in this case

- leads to defense strategy of randomization
  - make it more difficult for an attacker to guess behavior in an “unspecified” state
  - for example, Windows OS now includes address-space layout randomization
Jailbreak Attack (simplified example)

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>EAX = “whitehouse”</td>
</tr>
<tr>
<td>100004</td>
<td>JSR open_connection</td>
</tr>
<tr>
<td>100008</td>
<td>...</td>
</tr>
<tr>
<td>10000C</td>
<td>stack ptr</td>
</tr>
<tr>
<td>100010</td>
<td>100000</td>
</tr>
<tr>
<td>100014</td>
<td>stack ptr</td>
</tr>
<tr>
<td>100018</td>
<td>100000</td>
</tr>
<tr>
<td>10001C</td>
<td>stack ptr</td>
</tr>
<tr>
<td>100020</td>
<td>100000</td>
</tr>
<tr>
<td>100024</td>
<td>100000</td>
</tr>
<tr>
<td>100028</td>
<td>100000</td>
</tr>
<tr>
<td>10002C</td>
<td>100000</td>
</tr>
<tr>
<td>100030</td>
<td>100000</td>
</tr>
</tbody>
</table>

Code with vulnerability

- Read(buffer)
- RETURN
Jailbreak Attack (simplified example)

Stack:

100000: EAX = “whitehouse”
100004: JSR open_connection
100008: ...
100014: stack ptr
100018: 100000
10001C: 10000C
100020: 100020
100024: 100024
100028: 100028
10002C: 10002C
100030: RETURN

Code with vulnerability:

Read(buffer)
Jailbreak Attack (simplified example)

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>EAX = &quot;whitehouse&quot;</td>
</tr>
<tr>
<td>100004</td>
<td>JSR open_connection</td>
</tr>
<tr>
<td>100008</td>
<td>...</td>
</tr>
<tr>
<td>10000C</td>
<td>100000</td>
</tr>
<tr>
<td>100010</td>
<td>stack ptr</td>
</tr>
<tr>
<td>100014</td>
<td></td>
</tr>
<tr>
<td>100018</td>
<td></td>
</tr>
<tr>
<td>10001C</td>
<td></td>
</tr>
<tr>
<td>100020</td>
<td></td>
</tr>
<tr>
<td>100024</td>
<td></td>
</tr>
<tr>
<td>100028</td>
<td></td>
</tr>
<tr>
<td>10002C</td>
<td></td>
</tr>
<tr>
<td>100030</td>
<td></td>
</tr>
<tr>
<td>100034</td>
<td></td>
</tr>
</tbody>
</table>

code with vulnerability
Read(buffer)
RETURN
Beyond Simple Randomization

- combine randomization with parallelism and checkpointing:
  - generate several slightly different versions of the same program
  - run these versions in lockstep in parallel on different cores
  - look for discrepancies in behavior

- choose randomization parameters in a way that “symmetric” attacks become difficult

* research funded by IARPA under Program Manager Dr. Carl Landwehr
Example: Stack Addressing
MULTI-VARIANT PARALLEL PROGRAM EXECUTION TO DETECT MALICIOUS CODE INJECTION

Inventors: Michael Franz, Irvine, CA (US);
Andreas Gal, Irvine, CA (US); Babak Salamat, Irvine, CA (US)

Assignee: The Regents of the University of California, Oakland, CA (US)

Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 1163 days.

Appl. No.: 12/075,127
Filed: Mar. 7, 2008

Field of Classification Search ........................................... 714/127
See application file for complete search history.

References Cited

2003/0145253 A1 7/2003 de Bonet

2003/0145253 A1 7/2003 de Bonet

FOREIGN PATENT DOCUMENTS

OTHER PUBLICATIONS

(Continued)
Beyond Randomization

- most programs have deterministic behavior even when they operate outside of their specification; this is exploited by attackers.

- randomization makes it harder for an attacker to guess “out of specification” behavior, but randomization parameters can still be discovered.

- running several different randomized versions in parallel sets the threshold much higher: each version must be individually subverted without collateral damage affecting the other versions.
Technology Context

- multi-core processors are rapidly becoming the norm, even in cellphones (!)

- but: most day-to-day computer programs are inherently sequential and cannot (yet) be parallelized to make full use of this hardware

- use the extra processing cores for enhancing security, rather than performance
Checkpointing

- different program versions running in lockstep are expected to perform semantically equivalent operations at the same time

- possible checkpointing granularities:
  - system calls: all versions must be seeing same call with same parameters
  - note that a program can cause harm only through system calls
  - instruction-level: must be graduating instructions with same opcode across all participating cores
Hardware Support

Core 1

fetch
decode
...
execute
...
retire

Core 2

fetch
decode
...
execute
...
retire

LD -8(R6), R2
LD -12(R6), R3
ADD R2, R3
ST R3, -20(R6)
MULI #10, R3
ST R3, -24(R6)

LD 12(R2), R1
LD 18(R2), R5
ADD R1, R5
ST R5, 32(R2)
MULI #10, R5
ST R5, 40(R2)
More Hardware Support

- add transactional memory features
  - execute program in “chunks”
  - checkpoint a chunk across cores, commit to memory only when state identical
  - roll back if no agreement

- combine with majority voting when more than 2 cores are available (“RAID for computations”)
  - roll back corrupted versions
  - synthesize correct state from surviving versions
An Even Bolder Approach

"create a unique version of every program for every person in the universe"

* research funded by DARPA I2O under Program Managers Dr. Mari Maeda, Dr. Howie Shrobe, and Dr. Robert Laddaga
no change in process for developers

all the magic happens transparently “in the cloud” (organizations may choose to do this internally)

no change in process for users

subsequent downloaders receive functionally identical but internally different versions of the same software
critical factor: time interval between when a patch is available and when it is applied

the highest-value users often turn out to be the most vulnerable (international business travellers who cannot download patches while roaming, military deployed in areas with low connectivity)
“biologically inspired” software diversity is not a new idea

lots of previous work in the fault tolerance community, dating back to the 1970’s

n-version systems in software engineering research, found surprising amounts of coincident failures of supposedly independent software versions (1980’s)

Forrest’s research on compiler-guided code variance (1990’s), limited to “peephole” optimizations, parametrization (insufficient resources to perform true start-to-finish compilation over and over)

randomization techniques in operating systems (2000’s)
What’s The Cost?

- run-time cost
  - current unicompilers focus on finding the “best” of several alternative code paths, using heuristics
  - a multicompiler enumerates all the alternative paths that implement the same semantics and then gives a different alternative to each successive end-user
  - the difference between the “best” path and a semantically equivalent alternative path chosen for the sake of implementation diversity represents a potential performance loss
  - there are often many alternative paths that all have the “best” performance
  - hardware evolution keeps diminishing the performance differential between “best” path and “sub-optimal” paths

- up-front cost for generating versions
Threat Model

- Arbitrary Code Execution (ACE)
  - code injection
  - return-to-libc
  - return-oriented programming
  - jump-oriented programming
  - all are Turing complete with standard libraries

- Advanced Persistent Threats
  - adversaries who have time, money, and resources
  - possibly nation-states

- compiler is open-source

- adversaries have better resources and may even be smarter than us
Transformations

- remove the **vector**
  - permute stack objects
  - insert random pad

- render the **payload** ineffective, especially focusing on ROP
  - instruction scheduling
  - equivalent instruction selection
  - NOP insertion

- add additional **whole-program diversity** at link-time
  - base address randomization
  - function permutation
Existing Defenses

- non-executable memory protections such as Microsoft's Data Execution Prevention (DEP), CPU-supported non-executable memory (NX/XD), and mandatory code-signing such as on iPhone OS
  - attacker cannot inject new code

- as it turns out, this does not stop the modern attacker, who can devise an attack that **indirectly** uses **perfectly legitimate code already present on the target computer**
  - “return oriented” attack (aka “borrowed code”)

- it turns out to be extremely difficult to defend against this indirect type of attack using conventional means
  - “Return-Less Kernel” and “Gadget-Free Binaries” approaches defeated within months, see Schacham’s “Return-Oriented Programming Without Returns”
Return-Oriented Programming

- any reasonably large code base includes a Turing-complete set of “Gadgets” that can be used to splice together arbitrary programs (including branching and looping)
  - compounded on x86 by variable length instructions

- massive-scale compiler-generated code diversity prevents the attacker from compiling a “Gadget Dictionary” that is identical across many machines

- in conjunction with the non-executable stack hardware feature, this is a second-order defense
  - even if the attacker were able to overwrite the stack, a return oriented attack would then require detailed knowledge (down to individual bytes!) of the installed binaries on the target
  - we can prevent such a knowledge from existing
ROP Attack (simplified example)

- Code with vulnerability
  - Read(buffer)
  - RETURN

Legitimate code on target machine

- in Display_ColorText
  - ...555A00 EAX = “black”
  - 555A04 else
  - 555A08 EAX = “white”
  - 555A0C RETURN

- in HelpTexts_IOSpecific
  - ...212500 EBX = “keyboard”
  - 212504 else
  - 212508 EBX = “mouse”
  - 21250C RETURN

- in Audio_LowerVolume
  - ...ABCD00 SUBI #5000, @EBX
  - ABCD04 RETURN

- in FileSystem_QualifiedDirectoryName
  - ...777700 JSR strg_concat_EAX_EBX
  - 777704 RETURN

- in PrintManager_Prepare
  - ...919100 JSR open_connection
  - 919104 RETURN
ROP Attack (simplified example)

in Display_ColorText

... 555A00 EAX = “black”
     555A04 else
     555A08 EAX = “white”
     555A0C RETURN

in HelpTexts_IOSpecific

... 212500 EBX = “keyboard”
     212504 else
     212508 EBX = “mouse”
     21250C RETURN

in Audio_LowerVolume

... ABCD00 SUBI #5000, @EBX
     ABCD04 RETURN

in PrintManager_Prepare

... 919100 JSR open_connection
     919104 RETURN

in FileSystem_QualifiedDirectoryName

... 777700 JSR strg_concat_EAX_EBX
     777704 RETURN

EAX

EBX

Stack

stack ptr

code with vulnerability

Read(buffer)

RETURN
ROP Attack (simplified example)

Stack

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
...
555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific
...
212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume
...
ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare
...
919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName
...
777700 JSR strg_concat_EAX_EBX
777704 RETURN

Stack
EAX
white
EBX
ROP Attack (simplified example)

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
... 555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific
... 212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume
... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare
... 919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName
... 777700 JSR strg_concat_EAX_EBX
777704 RETURN
ROP Attack (simplified example)

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
...
555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific
...
212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume
...
ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare
...
919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName
...
777700 JSR strg_concat_EAX_EBX
777704 RETURN
### ROP Attack (simplified example)

#### Code with vulnerability

```assembly
Read(buffer)
RETURN
```

#### in Display_ColorText

```assembly
... 555A00 EAX = "black"
555A04 else
555A08 EAX = "white"
555A0C RETURN
```

#### in HelpTexts_IOSpecific

```assembly
... 212500 EBX = "keyboard"
212504 else
212508 EBX = "mouse"
21250C RETURN
```

#### in Audio_LowerVolume

```assembly
... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN
```

#### in PrintManager_Prepare

```assembly
... 919100 JSR open_connection
919104 RETURN
```

#### in FileSystem_QualifiedDirectoryName

```assembly
... 777700 JSR strg_concat_EAX_EBX
777704 RETURN
```
ROP Attack
(simplified example)

Stack

EAX
white

EBX
mouse

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
...
555A00    EAX = “black”
555A04    else
555A08    EAX = “white”
555A0C    RETURN

in HelpTexts_IOSpecific
...
212500    EBX = “keyboard”
212504    else
212508    EBX = “mouse”
21250C    RETURN

in Audio_LowerVolume
...
ABCD00    SUBI #5000, @EBX
ABCD04    RETURN

in FileSystem_QualifiedDirectoryName
...
777700    JSR strg_concat_EAX_EBX
777704    RETURN

in PrintManager_Prepare
...
919100    JSR open_connection
919104    RETURN
ROP Attack (simplified example)

Stack

EAX
- white

EBX
- house

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
...
555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific
...
212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume
...
ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare
...
919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName
...
777700 JSR strg_concat_EAX_EBX
777704 RETURN
ROP Attack (simplified example)

Stack

- EAX: white
- EBX: house
- Subroutine addresses:
  - 555A08: EAX = “black”
  - 212508: else
  - 555A0C: RETURN

in Display_ColorText

- 555A00: EAX = “black”
- 555A04: else
- 555A08: EAX = “white”
- 555A0C: RETURN

in HelpTexts_IOSpecific

- 212500: EBX = “keyboard”
- 212504: else
- 212508: EBX = “mouse”
- 21250C: RETURN

in Audio_LowerVolume

- ABCD00: SUBI #5000, @EBX
- ABCD04: RETURN

in PrintManager_Prepare

- 919100: JSR open_connection
- 919104: RETURN

in FileSystem_QualifiedDirectoryName

- 777700: JSR strg_concat_EAX_EBX
- 777704: RETURN

Read(buffer)
RETURN

code with vulnerability

stack ptr
ROP Attack (simplified example)

Code with vulnerability

in Display_ColorText

... 555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific

... 212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume

... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare

... 919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName

... 777700 JSR strg_concat_EAX_EBX
777704 RETURN

Stack

EAX

whitehouse

EBX

house

Read(buffer)
RETURN

stack ptr

100000
100004
100008
10000C
100010
100014
100018
100020
100024
100028
10002C
100030
100034
100038
10003C
100040
100044
100048
10004C
100050
100054
100058
10005C
100060
100064
ROP Attack (simplified example)

- **Display_ColorText**
  - `EAX = "black"
  - `EAX = "white"

- **HelpTexts_IOSpecific**
  - `EBX = "keyboard"
  - `EBX = "mouse"

- **Audio_LowerVolume**
  - `SUBI #5000, @EBX

- **PrintManager_Prepare**
  - `JSR open_connection

- **FileSystem_QualifiedDirectoryName**
  - `JSR strg_concat_EAX_EBX

- **Stack**
  - `EAX = 100000
  - `EBX = 100004

- **Code with vulnerability**
  - `Read(buffer)
  - `RETURN

- **Stack Pointer**
Return-Oriented Programming

- ROP is a “surgical” attack – it requires precise knowledge of the code in a program
  - Attackers can study their own copies of software to create attacks, or try a combination of brute force and binary exfiltration (Blind ROP)
  - Gadgets must be in the same location and be functionally equivalent to be “useful”

- Should we care?
  - ROP is actively used to exploit real systems, eg. Adobe Reader on fully patched Windows

- Defense: randomize in the compiler, after optimization

- Objective: attacker would need to exfiltrate a large binary in order to obtain a useful gadget dictionary
A second shift can be seen in the increasing number of use-after-free vulnerabilities that have been exploited. This vulnerability class includes issues that arise because of incorrect management of object lifetimes. One reason for this increase is that client-side vulnerabilities have become a prime focus for attackers, and object lifetime issues are a common vulnerability class encountered in applications.

Exploits that involve unsafe dynamic-link libraries (DLLs) were seen in a small percentage of cases from 2009 to 2012, but not in 2013. The introduction of technologies such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) has also affected the way attackers attempt to exploit vulnerabilities.

Figure 4 shows the techniques used in exploits targeting vulnerabilities in Microsoft products that were discovered over the past two years.

Source: Microsoft Security Intelligence Report, Vol. 16
Return-Oriented Programming

- unfortunately x86 has variable-length instructions and x86 branches can terminate at any byte location
- hence, there may be gadgets that the compiler never intentionally put there in the first place as valid instructions
- makes it very difficult to produce “gadget-free” code

Gadget: ADC [ECX], EAX RET
MOV [ECX], EDX

... 89 11 01 c3 ...
What Gadgets?

- want to assess whether a binary contains a Turing-complete set of Gadgets that an attacker could misuse

- built a model for such a minimum set of required Gadgets that we call the “MicroGadgets” set
  - any binary that contains the set is vulnerable

- note that a binary that doesn’t contain our set may still be vulnerable, because our model is just one of many sets of Gadgets that achieve Turing-completeness
## Turing-Complete Gadget Sets

<table>
<thead>
<tr>
<th></th>
<th>Cent OS</th>
<th>Open SUSE</th>
<th>PCLOS</th>
<th>Fedora</th>
<th>Kubuntu 7</th>
<th>Kubuntu 11</th>
<th>Ubuntu 9</th>
<th>Ubuntu 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdb</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inkscape</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>qemu</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtuoso</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>php</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>samba</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>lyx</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>mplayer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gimp</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>mysql</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>doxygen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Defenses

- how do we defend against return-oriented attacks?
- focusing on four techniques
  - NOP Insertion – shift Gadgets
  - Equivalent Instruction Insertion – change Gadgets
  - Instruction Scheduling – remove or change Gadgets
  - Booby Traps
- additionally, want to make the diversification both reproducible (may need to send an error report to the developer) and tunable (controllable cost-benefit knob)
  - solution: during compilation, continuously flip a coin to choose among equivalent options
  - use tunable parameters to bias the coin
## NOP Insertion – Shifting Code

<table>
<thead>
<tr>
<th>Before Diversification</th>
<th>After NOP Insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gadget:</strong> ADC [ECX], EAX</td>
<td><strong>Gadget:</strong> Removed</td>
</tr>
<tr>
<td>MOV [ECX], EDX</td>
<td>MOV [ECX], EDX</td>
</tr>
<tr>
<td>ADD EBX, EAX</td>
<td>NOP</td>
</tr>
<tr>
<td>RET</td>
<td>ADD EBX, EAX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Diversification Code</th>
<th>After NOP Insertion Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 11 01 c3 ...</td>
<td>89 11 90 01 c3 ...</td>
</tr>
</tbody>
</table>
Equivalent Instructions – Changing Expected Behavior

Before Diversification

POP ECX
MOV [ECX], EDX
ADD EBX, EAX
ROL [ECX], 1
RET

After Substitution

POP ECX
LEA EAX, [EDX]
ADD EBX, EAX
ADD AL, [ECX]
RET

Gadget: Modified
Instruction Scheduling – Rearranging Code

Before Diversification

<table>
<thead>
<tr>
<th>POP ECX</th>
<th>MOV [ECX], EDX</th>
<th>ADD EBX, EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>89 11</td>
<td>01 c3</td>
</tr>
</tbody>
</table>

Gadget: Removed

After Scheduling

<table>
<thead>
<tr>
<th>POP ECX</th>
<th>ADD EBX, EAX</th>
<th>MOV [ECX], EDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>01 c3</td>
<td>89 11</td>
</tr>
</tbody>
</table>

Gadget: Removed
Dynamic Control Flow Diversity

- basic idea: instead of running one version of a program, rapidly switch between different versions of the same program
  - replicate functions or basic blocks
  - diversify each replica with transformations which affect side-channel characteristics
  - during execution, randomly switch between available versions, potentially on every control flow edge
- as a result, no two executions of the same program follow the same path, even when the same inputs are used
- effective for automatically mitigating side channel attacks [see our NDSS 2015 paper]
- also see our second NDSS 2015 paper on combining code diversity with Control Flow Integrity
Control Flow Diversity

main()
a()
b()
c()
d()

main()
a()
b()
c()
d'()
Control Flow Diversity

Program Trace

main() → a() → d() → b() → d()

timing channel

static
Variant 1

static
Variant 2
dynamic
Variant
Adversaries React to Diversity

- diversity relies on the fact that adversaries don’t know the precise code layout on the target

- recent attacks have adapted by either
  - exfiltrating the whole binary for off-line analysis or by
  - searching for code gadgets directly on the target

- solutions:
  - in addition to making the stack non-executable, we need to make the code segment non-readable
  - [see our upcoming paper at IEEE S&P 2015 (“Oakland”)]
Active Defenses: Booby Traps

- “don’t just sit there, do something about it”

- currently, there is little disincentive against brute-force attacks

- defenders not making good use of **home field advantage**: knowing your own territory [hopefully] better than the adversary

- analogy in the physical world:
  - booby traps, landmines, tripwires – placed by a defender to slow down and discourage an aggressor
Diversification Moves Gadgets

Before diversification

1 3 2 4

After diversification

1 3 2 4

- an attack requires a specific set of gadgets, and the attacker needs to be able to locate them

- so attackers “guessing” gadget locations need to do a pretty good job to begin with, because they need to guess the locations for all of their required gadgets fairly accurately
Booby Traps provide active defenses, substituting “logic bombs” in the locations of some of the moved gadgets.

An attack requires multiple gadgets; defensive objective is satisfied if the attacker triggers just a single booby trap.
Booby Trapping Code

- Booby Traps are “woven” into programs
  - completely automatic
  - completely transparent to the developer of the program being fortified in this manner
  - control flow normally never reaches the code that is woven into the program in this manner
  - once woven together, the original program and the extra defensive code become inseparable

- there is no runtime overhead (other than possible cache effects due to code growth) because the extra code is never executed unless an attacker triggers it

- not all gadgets are equally valuable – can increase Booby Trap density near such “valuable” gadgets
Possible Responses

Various responses possible when a booby trap is triggered:

- **Recovery** - restore known good program state and resume operation (operate through attacks)
- **Version Flux** - replace program with a different version/variant
- **Honeypot / Forensics** - learn more information about the attack, trick attacker into thinking attack was successful
- **Counterattack** - strike back at attacker
- **Situational Awareness** - broadcast attack information to other parties

[see our paper at 2013 New Security Paradigms Workshop]
触发陷阱会将控制权返回给防守者。
- 如果这段代码被执行，我们知道发生了不好的事情。
- 我们只创建“管道”在这里——可以插入任意策略，当陷阱被触发时应该做什么。

```assembly
225d  pop  r14
225f  pop  rbp
2260  pop  rbx
2261  ret
```

```assembly
225b  jmp  2275
225d  nop
225e  nop
226f  nop
2270  jmp  BOOBY_TRAP
2275  pop  r14
2277  pop  rbp
2278  pop  rbx
2279  ret
```
Implementation

- created a diversifying compiler on the LLVM Compiler Infrastructure
  - migrated from 2.8 to 2.9 and maintaining 3.0 implementation
  - prototype GCC 4.6.2 with NOP insertion
  - link time transformations require updated linker

- all transformations are compatible with each other
  - major advantage over other techniques that artificially create diversity

- complete source code compatibility
  - all transformations are done in the backend or linker

- complete binary compatibility
  - diversified software runs alongside conventional programs
Implementation

- **deterministic output**
  - enables developers to determine easily reproduce an executable, given random number seed

- **almost secretless**
  - seed is *not* embedded into the resulting binary
  - seed is used for the entire build
  - major effort to prevent leakage of information
    - cryptographically safe pseudo-random number generator based on AES
    - cannot easily “fingerprint” just part of a binary to gain knowledge about diversification in the whole binary
Random Seed is The Only Secret

Apps evolve over time

v1.0  v1.1  v1.3

Diversification Engine

random seeds drive diversification, can be
- given to clients
- stored in cloud
- distributed using secret sharing

different variants of same App version
Applications

- can now protect any existing software that compiles with GCC or LLVM
  - have used Mozilla’s Firefox and Google’s Chromium browser, Python, and Apache as our model programs
- built entire Linux system with diversification
  - used Linux From Scratch and a diversifying GCC
- using newly invented compiler techniques, were able to reduce overhead to 1% without a significant security impact [see CGO 2013 paper]
Take-Away Points

- industrial strength implementation can right now diversify
  - entire Linux distribution (Linux from Scratch)
  - Mozilla’s Firefox and Google’s Chromium browsers
  - Python, Apache

- cost of diversifying Firefox (33 Mloc) is sub $0.01 per unique copy at retail pricing on Amazon’s cloud

- with profile-guided optimization, performance overhead can be pushed down to 1% at good security settings

- by adding dynamic diversity, we can even overcome traditionally “difficult” problems such as side-channel information leakage
Thank You