Compilers, Memory Errors and Hardening Techniques

Advanced Compiler Design, 18.5.2016

Antonio Hüseyin Barresi
Co-founder & CEO
xorlab AG
Executive Opinion Survey 2015

Cyber attack is seen as the greatest global risk to doing business by executives in eight countries…

Source: World Economic Forum
Executive Opinion Survey 2015

What does that have to do with...

Compilers?
This talk

• Cyber attacks and compilers
• Memory errors and exploitation
• Hardening techniques and run-time security
About

Antonio Barresi
antonio.barresi@xorlab.com
@AntonioHBarresi

Co-founder of
xorlab
https://www.xorlab.com
Cyber attacks cost the world economy $400 billion yearly.

Cyber crime is a growth industry
“Cybercrime is a tax on innovation and slows the pace of global innovation”

Direct financial losses, loss of competitiveness and other impact

High-income countries suffer the most

- 0.9% of GDP in average
- Loss of 5.8 Bn CHF (CH 2013, GDP 646 Bn CHF)
Example
100 banks, 30 countries, 2 years

$1 Billion
The great bank robbery

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as Carbanak.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
The great bank robbery

An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as Carbanak.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
The great bank robbery

An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as Carbanak.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
The great bank robbery

An analysis of the campaign has revealed that the initial infections were achieved using spear phishing emails that appeared to be legitimate banking communications, with Microsoft Word 97 – 2003 (.doc) and Control Panel Applet (.CPL) files attached. We believe that the attackers also redirected to exploit kits website traffic that related to financial activity.

The email attachments exploit vulnerabilities in Microsoft Office 2003, 2007 and 2010 (CVE-2012-0158 and CVE-2013-3906) and Microsoft Word (CVE-2014-1761). Once the vulnerability is successfully exploited, the shellcode decrypts and executes the backdoor known as Carbanak.

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
The great bank robbery

„CVE-2012-0158 is a buffer overflow Vulnerability in the ListView / TreeView ActiveX controls in the MSCOMCTL.OCX library."

https://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/
Exploit and Zero-Day Attack

Exploit = “An **exploit** is a **piece of software, a chunk of data**, or a sequence of commands that **takes advantage of a bug**, glitch or vulnerability in order to...“

Zero-Day Attack = “A **zero-day** attack or threat is an attack that exploits a **previously unknown vulnerability** in a computer application, ...“

http://en.wikipedia.org/wiki/Exploit_(computer_security)
http://en.wikipedia.org/wiki/Zero-day_attack
Software attacks

• Attacking software is an important capability for attackers
  • Scales well
  • Can be done remotely
  • Very powerful
Arbitrary code-execution
Goal of an attacker
Most important thing

Profitability
Profit

Revenue –
(Costs + expenses + taxes)
Cyber criminals

• Have *costs* and *expenses* too!

• An economic decision

• Fighting cyber crime means limiting its *profitability*
Cyber criminals

- **Have costs and expenses too!**
- **An economic decision**
- **Fighting cyber crime means limiting it’s profitability**
Software hardening

• Compile and run-time techniques

• To make software exploitation harder

• But also slower
Exploit mitigations since the 90s

- The Anderson Report
- Morris Worm
- StackGuard
- ret2libc
- Smashing the stack
- PaX
- PAGEEXEC
- Advanced ret2libc
- PaX ASLR
- /GS
- /SAFESEH
- SEH Overwrite
- Heap spraying in browsers
- ASLR
- Linux
- DEP
- Lin + Win
- XP SP2
- /GS
- /SAFESEH
- ROP
- ASLR
- Windows Vista
- Integrity levels
- Windows Vista
- Nozzle
- Heap spraying protection
- JOP
- COP
- vtguard
- Win8 + IE 10
- ROPGuard
- Win8 «Sandbox»
- Win32k lockdown + App Container
- «Sandbox»
- Win32k
- lockdown + App Container
- Fully enforced ASLR
- Nozzle
- Heap spraying protection
- EMET 5.2
- Control Flow Guard
- VS15 + Win 10 or 8.1U3

Timeline:
- 1972
- 1988
- 1996
- 2000
- 2001
- 2002
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
Attacker Return

\[
\text{Attacker Return} = \left( \frac{\text{Gains per use}}{\text{Opportunities to use}} \right) \times \left( \frac{\text{Cost to acquire vulnerability}}{\text{Cost to weaponize}} \right)
\]
Attacker Return

\[
\text{Attacker Return} = \left( \frac{\text{Gains per use}}{\text{Opportunities to use}} \times \right) - \left( \frac{\text{Cost to acquire vulnerability}}{\text{Cost to weaponize}} \right) + \]

Increase costs

Mitigating Software Vulnerabilities, July 2011, Microsoft
Hardening value chain

Source → Compiler & Linker → Binary

Application
Run-time libraries
Operating system
Hardware
Hardening value chain

- **Source**: Compiler & Linker
  - Compiler optimizations
  - Stack canaries
  - SAFESEH
  - /GS
  - Compiler warnings

- **Binary**: Compiler & Linker
  - Pointer obfuscation

- **Run-time libraries**: Application
  - Hardened heap
  - ROP mitigations (EMET)
  - SEHOP
  - Sandboxing

- **Operating system**: Hardened heap
  - DEP & ASLR

- **Hardware**: Hardened heap
  - GOT protection
  - NX-bit for DEP
  - DEP & ASLR
Software attacks
Attacks

Victim

Runs a vulnerable web browser or PDF reader

Exploits memory error within vulnerable victim software

Attacker

Runs a malicious web server serving HTML documents that trigger a memory error within the web browser

Sends a malicious PDF attachment by email

Runs an exploit that triggers a memory error within the web server software

GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=5r4q7ib3vtdq120183f8

$>./exploit 192.168.1.28

Runs a vulnerable web server software
Memory corruption

- Exist in programs written in “unsafe” languages
  - C/C++

- Two basic types exist
  - Spatial and temporal memory errors

- Allow attackers to corrupt memory in a more or less controllable way
«Unsafe» languages

Allow low-level access to memory
  • Typed pointers & pointer arithmetic
  • No automatic bounds checking / index checking

Weakly enforced typing
  • Cast (almost) anything to pointers

Explicit memory management
  • Like malloc() & free() in C
```c
#include <stdio.h>
#include <math.h>

long computation(int x, int y, int z, double f) {
    return (long)(((x*y)/z)*sin(f));
}

void main() {
    *(int*)computation(32, 64, 2, M_PI/6) = 128;
    return;
}
```

```
shell:~$ gcc -o segfault segfault.c -lm
shell:~$ ./segfault
Segmentation fault (core dumped)
shell:~$
```
#include <stdio.h>
#include <string.h>
#define STDIN 0

void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}

void main() {
    printf("read> ");
    vulnFunc();
    return;
}

shell:~$ gcc -o vuln vuln.c -fno-stack-protector
shell:~$ ./vuln
read> hi there!
shell:~$ ./vuln
read> AAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
shell:~$
Types of memory errors

Spatial error
- De-reference pointer that is out of bounds

Temporal error
- De-reference pointer to freed memory
Exploiting memory errors

**Spatial error**

- Overwrite data or pointers
- Used or de-referenced later

**Temporal error**

- Make application allocate memory in the freed area
- Used as old type
Attackers use memory errors to

Overwrite data or pointers
  • That might be used to overwrite data or pointers
  • Function pointers, sensitive data, index values, control-flow sensitive data etc.

Leak information
  • E.g., corrupt a length field

Construct attacker primitives
  • Write primitive (write any value to arbitrary address)
  • Read primitive (read from any address)
Types of bugs

- Out-of-bounds bugs / Buffer overflows
  - On stack or heap
- Dangling pointer / Use-after-free
- Integer bugs, signedness bugs
- Format string bugs
- Uninitialized memory
- NULL pointer dereference
Memory corruption attack

**Write primitive**

*0xe8a0f000 = 0x41414141*
Attack types

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

Attack model according to: “SoK: Eternal War in Memory” Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
Control-flow hijack attack

- Most powerful attack
- Hijack control-flow
  - To attacker supplied arbitrary machine code
  - To existing code (code-reuse attack)
- Corrupt code pointers
  - Return addresses, function pointers, vtable entries, exception handlers, jmp_bufs
Control-flow hijack attack

Most ISAs support indirect branch instructions

E.g., x86 ret, indirect jmp, indirect call

branch *fptr
Control-flow hijack attack

fptr corrupted by attacker

branch *fptr

Attacker goal: hijack control-flow to injected machine code or to “evil functions“
Control-flow hijack attack

- Indirect call to func()
- Hijacked indirect call
Non-eXecutable data (NX)

Make data regions non-executable (by default)

Changing protection flags or allocating rwx memory still possible

Required for JITs
Non-eXecutable data (NX)

Code regions will be non-writable

• Else code corruption attacks possible

Also known as

• Data Execution Prevention (DEP) on Windows

• NX, Non-eXecutable Memory on Linux

• $W \wedge X$, on OpenBSD

Implemented by hardware where available (NX-bit)
Binary images need to provide separate sections / segments that can be mapped exclusively as

- rw- OR r-x
- Linker support required

Self-modifying code not allowed

- Compiler support required
- If code is generated just-in-time, explicit rwx allocation required
Bypassing NX / DEP

Only use existing code

Code-reuse attack

- ret2libc, ret2bin, ret2*
- Return-oriented programming (ROP)
- Jump/Call-oriented programming

Use code-reuse technique to change protection flags

Make memory executable

- mprotect/VirtualProtect
- mmap/VirtualAlloc
Stack-based buffer overflow

%eip and %ebp under control

Local variables and buffers under attacker control
Code-reuse attack

Before NX/DEP

Return to attacker supplied code
→ Shellcode

Use existing code

Executable or libraries
E.g., mprotect()

Stack after read()
Let's call mprotect()

```
mprotect(address_shellcode, 4096, 0x1|0x2|0x4)
```

0x1 | 0x2 | 0x4 = RWX

mprotect() will make the stack executable

...and return to our shellcode
Code-reuse attack

After `mprotect()` our stack is executable again

`mprotect()` will return to our shellcode

x64 or ARM:

Fill registers with parameters before invoking `mprotect()`
Return-oriented programming

Use available code snippets ending with ret instruction

Called gadgets / ROP chain
E.g., write primitive

```
1 pop %edx; ret;
2 pop %eax; pop %ebx; ret;
3 mov %edx, (%eax);
   mov $0x0, %eax;
   ret;
```

```
address gadget4
address gadget3
dummy value
  address
dummy gadget2
  value
dummy gadget1
dummy ebp
buf[1024]
```

Stack

%esp

Code

%ebp

rw-

r-x
Return-oriented programming

• Turing complete although not required

• Can be initiated over call or jmp as well

• Need to be in control of memory %esp is pointing to
  Or make %esp point to area under control

• Also possible with jmp or call gadgets
Addresses in memory

To hijack control-flow or to corrupt memory an attacker needs to know where things are in memory

- Addresses of data or pointers to corrupt
- Addresses of injected shellcode/payload
- Addresses of gadgets
Addresses in memory

Once upon a time...

Addresses were more or less predictable

Executables and libraries were prelinked to certain addresses

Stack and Heap base addresses were fixed
  • With differences at runtime for specific locations due to the dynamic behaviour of the process
ASLR

Today most Operating Systems implement Address Space Layout Randomization

Randomize memory layout of processes to make address prediction or guessing hard

What can be randomized?

- OS: Stack, heap and memory mapping base addresses
- OS, compiler, linker: Executables and libraries
  - Position-independent or relocatable code
Randomization of layout

Use source of randomness to randomize within ranges
ASLR – implementation issues

In general: Usage of fixed addresses not allowed

Code should be position-independent or relocatable

Linux/ELF:
- PIC & PIE supported, libraries all PIC, executables sometimes still prelinked

Windows/PE:
- No PIC support! But libraries/executables relocatable!

x86 32bit PIE/PIC slower, no IP relative data addressing
Relocated PE images not sharable between processes
ASLR effectiveness

Enough entropy?

• Base range size
• Alignment constraints
• Address width (64bit is better than 32bit)

Randomization strategy

Per process, system-wide per boot

Source of randomness

ASLR only effective without exceptions
Bypassing ASLR

Low entropy

- Brute-force addresses (multiple attempts required)

Memory leaks (information disclosure)

- Leak addresses to derive base addresses
- Construct and enforce a leak by memory corruption

Application and vulnerability specific attacks

- Force predictable memory state
  - Heap-spraying / Heap massaging
- Pointer inference
  - Use a side-channel
- Avoid using exact addresses
  - Only corrupt least significant bytes i.e. offsets
Memory leak

Runs web server software with memory leak and exploitable memory corruption vulnerability

1. Trigger memory leak

2. Parse response for leaked memory and construct exploit

3. Exploit memory error with tailored exploit
Memory leak

mprotect = leaked pointer – static offset

0x0ebb0880 = 0x0efa4604 - 0x003f3d84
void memLeak() {
    char buf[256];
    scanf("%s", buf);
    printf(buf);
}

void main() {
    printf("echo> ");
    memLeak();
    printf("\n");
    return;
}

shell:~$ gcc -o memleak memleak.c
memleak.c: In function 'memLeak':
memleak.c:9:2: warning: format not a string literal and no format arguments [-Wformat-security]
shell:~$ ./memleak
echo> hi
hi
shell:~$ ./memleak
echo> %llx,%llx,%llx,%llx,%llx,%llx,%llx
1,7f1f517cac0.a,ffffffff,0,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c
shell:~$
Heartbleed – CVE-2014-0160 - OpenSSL

http://en.wikipedia.org/wiki/Heartbleed
Simplified Heartbleed explanation by FenixFeather licensed under CC
Constructed arbitrary read
CVE-2011-2371 – Firefox vulnerability

```c
// this returns a string at absolute address addr_to_disclose with
// length numberOfBytes

function arbitrary_leak_string(addr_to_disclose, numberOfBytes) {

  array_before_slot_of_infoDisclosure[126] = addr_of_infoDisclosure_slot-8-8-512;
  // NOTE: addr_of_InfoDisclosure_slot-8-8 only works if the heap massage worked out i.e.
  // the infoDisclosure reallocated slot is adjacent to the other allocations of 512 bytes
  // i.e. attacker controlled memory, in our case we have something in-between which requires
  // us to adapt this value so that the struct pointer points correctly to the flags & length
  // part of the string structure
  // SEEMS THAT THERE IS EXACTLY ONE 512 bytes BLOCK IN BETWEEN! -512 required!
  array_before_slot_of_infoDisclosure[127] = 0xffff0000; // type string
  // ptr to string structure [124]

  var tmpLen = numberOfBytes;
  if(tmpLen % 2 != 0) tmpLen += 1; // multiple of two
  tmpLen /= 2; // unicode is half of the length

  array_before_slot_of_infoDisclosure[124] = (tmpLen << 4 | 4); // length of unicode string
  array_before_slot_of_infoDisclosure[125] = addr_to_disclose;
```
DEP & ASLR

DEP & ASLR are generic defenses

Exploitation becomes harder for all vulnerability classes & attack techniques

Together quite effective
  • If implemented correctly

Injecting code and corrupting pointers with exact addresses is in general desirable for attackers
DEP & ASLR are not enough

A determined attacker will use code-reuse techniques and memory leaks to bypass DEP & ASLR.
Additional protections

Raise exploit development costs with additional protections

• The more line of defenses, the better!

• Implement protections against specific vulnerability classes and exploitation techniques
May require **source code changes** (annotations) and/or **recompilation** of the application

Compile-time and checks at run-time

- Stack canaries / cookies
- Pointer obfuscation
- /GS
- /SAFESEH (link-time, provide list of valid handlers)
- SEHOP (run-time, walk down SEH chain i.e. integrity check)
- Virtual Table Verification (VTV) & vtguard
- Control-Flow Guard
Stack canary / cookie

```c
void vulnFunc() {
    <copy canary>
    char buf[1024];
    read(STDIN, buf, 2048);
    <verify canary>
}
```
Mitigations deployment delay

• Deployment of exploit mitigations requires time
• Source or binary compatibility risk
• Support from multiple sides required

Attackers have time to adapt
Exploit mitigations since the 90s

- The Anderson Report
- Morris Worm
- StackGuard
- ret2libc
- Smashing the stack
- StackGuard
- ret2libc
- Advanced ret2libc
- PaX ASLR
- PaX PAGEEXEC
- /GS
- SEH Overwrite
- /SAFESEH
- Heap spraying in browsers
- ASLR
- Linux
- DEP
- Lin + Win
- XP SP2
- /GS
- /SAFESEH
- ROP
- ASLR
- Windows Vista
- Integrity levels
- Windows Vista
- Nozzle
- Heap spraying protection
- vtguard
- Win8 + IE10
- ROPGuard
- Win8
- «Sandbox»
- Win32k lockdown + App Container
- Control Flow Guard
- VS15 + Win10 or 8.1 U3
- Fully enforced ASLR
- EMET 5.2
Modern mitigations

• ROP mitigations (EMET)
• Control-Flow Integrity
  • Control-Flow Guard (CFG)
ROP mitigations (EMET)

- Run-time checks
- Detect ongoing ROP attacks
- Hook into sensitive APIs
Control-flow integrity (CFI)

• Original publication in 2005
  «Control-Flow Integrity – Principles, Implementations, and Applications» Abadi, Budiu, Erlingsson, Ligatti, CCS’05

• Many CFI implementations proposed since then
  • Compiler-based, binary-only (static rewriting)

• Check indirect control-flow transfers and limit the set of allowed targets
Control-flow integrity (CFI)

• CFI approaches

• “Fine-Grained Control-Flow Integrity through Binary Hardening”, DIMVA 2015
  Mathias Payer, Antonio Barresi, Thomas R. Gross

• “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity”, USENIX 2015
  Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, Thomas R. Gross
Control-flow guard (CFG)

- First adoption of a practical Control-Flow Integrity (CFI) implementation
- Requires recompilation and OS support
  - VS15 + Windows 10 or 8.1U3
- Restricts indirect call targets to a static global set of locations
Control-flow guard (CFG)

Guard CF Function Table
validFunc1
validFunc2
...

Compiler & Linker
Binary
Application
Run-time libraries

Control-flow guard (CFG)

Guard CF Function Table
validFunc1
validFunc2

Compiler & Linker

Binary

Application

Run-time libraries

Control-flow guard (CFG)

Guard CF Function Table
validFunc1 validFunc2

Compiler & Linker

Binary

Application

Run-time libraries

Control-flow guard (CFG)

Guard CF
Function Table
validFunc1
validFunc2

Compiler & Linker

Binary

Application
ntdll!LdrpValidateUserCallTarget

CFGBitmap
011010100101
100101011001
010101010100
101010101010
01110110....

Future?

• Hardening techniques make exploit development harder and increase the costs of an attack

• But there is no magic bullet

• Attacker’s still leverage memory leaks and code-reuse techniques to bypass all mitigations (ret2libc is 1997)
Pwn2Own 2016

Pwn2Own 2016: Chrome, Edge, and Safari hacked, $460,000 awarded in total

Pwn2Own 2016 in Numbers:
Total prizes awarded:
• Master of Pwn Points: 98
• Cash: US$ 460,000

Number of Attempts:
• Fully Successful: 7 (64%)
• Partially Successful: 1 (9%)
• Failed: 3 (27%)

Number of Successful Attempts Against:
• Apple Safari: 3/3 (100% Success)
• Microsoft Edge: 2/2 (100% Success)
• Adobe Flash: 4/5 (75% Success)
• Google Chrome: 5/2 (25% Success) NOTE: The actual vulnerability in Google Chrome had already been independently reported to Google; this is counted a partial success

Percentage of Successful or Partially Successful attacks that achieved SYSTEM or root privilege: 100%

Contestant Success Standings:
• Tencent Security Team Sniper (KeeuLab and PC Manager): 3/3 (100% Success)
• 360Vulcan Team: 1/5/2 (75% Success)
• JungHoon Lee (t0xihardt): 2/3 (68% Success)
• Tencent Security Team Shield (PC Manager and KeeuLab): 1/2 (50% Success)
• Tencent Xuanwu Lab: 0/1 (0% Success)

http://venturebeat.com/2016/03/18/pwn2own-2016-chrome-edge-and-safari-hacked-460k-awarded-in-total/
http://blog.trendmicro.com/pwn2own-day-2-event-wrap/
Conclusion
Conclusion

- Compilers help mitigate software attacks
- Software hardening increases costs of attacks
- Attackers still find ways to bypass all protections
Questions?

antonio.barresi@xorlab.com