CSE 40567 / 60567: Computer Security

Software Security 6
Homework #5 Released
Due: 3/28/17 at 11:59PM Eastern Time

See Assignments Page on the course website for details
Next Week: *We Steal Secrets*
In Class Screening + Discussion
What does metasploit do?

• Provides tools that make debugging, offset hunting and payload crafting easier

• Basic steps for exploiting a system via the framework:
  1. Choose and configure an exploit (over 900 available in the framework)
  2. Check whether a target system is vulnerable
  3. Choose and configure a payload
  4. Encode payload to evade IDS
  5. Execute the exploit
Disassembling code

• Broad knowledge of assembly language is essential for writing exploits
  ‣ Need to know where function calls exist in memory, the internal execution flow of the program, and the state of the heap / stack (eip / rip manipulation)
  ‣ Sometimes source code isn’t available; the binary must be examined in these cases
(gdb) disass main
Dump of assembler code for function main:
  0x00000000000400624 <+0>: push %rbp
  0x00000000000400625 <+1>: mov %rsp,%rbp
  0x00000000000400628 <+4>: sub $0x30,%rsp
  0x0000000000040062c <+8>: mov %edi,-0x24(%rbp)
  0x00000000000400633 <+11>: mov %rsi,-0x30(%rbp)
  0x00000000000400633 <+15>: cmpl $0x4,-0x24(%rbp)
  0x00000000000400637 <+19>: je 0x40063e <main+26>
  0x00000000000400639 <+21>: jmpq 0x400731 <main+269>
  0x0000000000040063e <+26>: mov $0x100,%edi
  0x00000000000400643 <+31>: callq 0x400520 <malloc@plt>
  0x00000000000400648 <+36>: mov %rax,-0x18(%rbp)
  0x0000000000040064c <+40>: mov $0x40082c,%eax
  0x00000000000400651 <+45>: mov -0x18(%rbp),%rdx
IDA PRO

https://www.hex-rays.com/index.shtml

https://youtu.be/vb18UVF4a_o
Fuzzing

- Black-box testing methodology
- Checks code modules for vulnerability to overflows
  - Many are not obvious to visual inspection
- Two fuzzing strategies are typically deployed
  1. Mutation-based fuzzers
  2. Generation-based fuzzers
CERT Basic Fuzzing Framework (BFF)

Mutational fuzzer

https://github.com/CERTCC-Vulnerability-Analysis/certfuzz

https://youtu.be/kSnc7Ri5ByA
w3af

http://w3af.org/

Features:

• Daemons
• Fast HTTP Client
• Output Manager
• Fuzzing Engine
• Knowledge base
skipfish

https://code.google.com/archive/p/skipfish/

Crawl results - click to expand:

Document type overview - click to expand:

application/xhtml+xml (53)

https://www.owasp.org/index.php/Automated_Audit_using_SKIPFISH
Protection Mechanisms Against Attack
Basic Strategies

- Check if known values in memory are being overwritten
- Have the compiler perform bounds checking
- Make it difficult for an attacker to find the necessary memory offsets
- Explicitly mark regions of memory as being non-executable

Philosophical question: can’t we solve this with good coding practices?
Canaries

**Goal:** detect that a return address has been altered before a function returns

**Implementation:** place a known value between a buffer and control data on the stack

**Three basic strategies:**

1. Terminator canaries
2. Random canaries
3. XOR canaries

Canary word next to return address

Frame 2

```
local  canary  sfp  ret  str
```

Frame 1

```
local  canary  sfp  ret  str
```

Top of Stack
Random canaries

• Random canary string chosen at program invocation
• Canary string is inserted into every stack frame
• To bypass the canary, the attacker needs to know what it is ahead of time

Operation:
1. Canary is verified before returning from function
2. If canary changed, exit program

Potential for DoS attack
Terminator canaries

Use a known canary: {0, newline, linefeed, EOF}

How do these characters help us?

• String functions will not copy beyond a terminator
  ‣ e.g., `strcpy()` returns when copying a null character

• Attacker can’t use string functions to corrupt the stack

Potential pitfall: the canary is known to the attacker
Random XOR canaries

Variation on the theme of a random canary: also consider control portions of the stack

Frame 1

Check that incorporates canary and ret

To attack, a matching XOR result must be computed from a new canary and ret address. This requires knowledge of the original canary and canary algorithm.
Stackguard

• The three canary strategies are implemented as a patch to gcc

• Minimal impact on performance
  ‣ Apache: 8% slower

• Caveat: canaries are not fool-proof
  ‣ Some buffer overflow attacks will leave the canaries intact
ProPolice (IBM)

- GCC 4.1: `-fstack-protector` and `-fstack-protector-all`
- Re-arrange stack layout to prevent pointer overflow

![Stack Layout Diagram]

- Protects args and local pointers from a buffer overflow
- Pointers, but no arrays

### String Growth
- Args
- Ret addr
- SFP
- CANARY

### Stack Growth
- Local string buffers
- Local non-buffer variables
- Copy of pointer args
-fstack-protector-strong (Google)

- Attempt to balance security and performance
- Protects more functions than -fstack-protector
  - Which protects less than 2% of functions
- But not as many as -fstack-protector-all
  - Which protects everything, whether needed or not

**Standard for some OSs:** OpenBSD, Hardened Gentoo

**Optional for others:** Debian, FreeBSD
Visual Studio’s /GS switch

Default compiler option since VS 2005

Added to all functions in VS 2010

Combination of ProPolice and random canary

If there is a “cookie” mismatch, the default behavior is to call \_exit(3)

___security_cookie = canary
Function prolog and epilog

**FuncFon prolog:**

```
sub esp, 8        // allocate 8 bytes for cookie
mov eax, DWORD PTR ___security_cookie
xor eax, esp     // xor cookie with current esp
mov DWORD PTR [esp+8], eax   // save in stack
```

**FuncFon epilog:**

```
mov ecx, DWORD PTR [esp+8]
xor ecx, esp

call @___security_check_cookie@4
add esp, 8
```
/GS stack frame

- Args
- Ret addr
- SFP
- Exception handlers
- CANARY
- Local string buffers
- Local non-buffer variables
- Copy of pointer args

String Growth

Stack Growth

Canary protects ret-addr and exception handler frame

Pointers, but no arrays
/GS is not foolproof

Evasion is possible: trigger exception before **canary** is checked

When an exception is thrown, the dispatcher walks up the exception list until the handler is found

- If no handler is found, the default is used

After overflow: handler points to attacker’s code

exception triggered $\implies$ control hijack
Additional VS flags

/SAFESEH: linker flag

- Linker produces a binary with a table of safe exception handlers
- System will not jump to an exception handler not on list
Additional VS flags

/SEHOP: platform defense (since Vista SP1)

- SEH attacks typically corrupt the “next” entry in the SEH list

- /SEHOP adds a dummy record at top of SEH list

- When exception occurs, the dispatcher walks up the list and verifies that the dummy record is there. If not, the process terminates.
If recompiling is not an option: Libsafe

http://directory.fsf.org/wiki/Libsafe

- Dynamically loaded library
- Intercepts calls to `strcpy(dest, src)`

Validates sufficient space in current stack frame:

```
|frame-pointer - dest| > strlen(src)
```

If so, does `strcpy()`. Otherwise terminates application.
Libsafe is not foolproof

Libsafe `strcpy()` can overwrite a pointer between `buf` and `sfp`. 
Compiler-based bounds checking

**Bounds Checking for C** (http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html)

- Every pointer expression derives a new pointer from a unique original pointer.
- Every pointer value is valid for just one allocated storage region.
- Pointer arithmetic expressions are checked for validity
- Implemented as patch to gcc
Compiler-based bounds checking

**SAFECode** (http://safecode.cs.illinois.edu/)

- Array bounds checking
- Loads and stores only access valid memory objects
- Type safety for a subset of memory objects proven to be type-safe
- Sound operational semantics in the face of dangling pointer errors
- Optional dangling pointer detection (induces overhead)
Compiler-based bounds checking

**AddressSanitizer** (https://github.com/google/sanitizers)
- Compiler instrumentation module (currently, an LLVM pass)

Can find:
Use after free (dangling pointer dereference), Heap buffer overflow, Stack buffer overflow, Global buffer overflow, Use after return, Initialization order bugs, Memory leaks

More on heap protection in a moment…
Tagging

• Tag the type of data in memory, and perform strong type checks during program execution

• Two key tags:
  1. memory is non-executable
  2. memory is non-allocated

Implemented via software or hardware (far more difficult to thwart)
No eXecute (NX) bit

Supported by Intel, AMD and ARM processors

x64 hardware page table entry
W xor X (OpenBSD)

Many bugs are exploitable because the address space contains memory that is both writeable and executable (permissions = $\overline{w} \land x$)

A serious hinderance would be to ensure no pages have $\overline{w} \land x$ permission

Implementation: use the NX bit to control permission at the hardware level on architectures that support it

As of 2015, userland and kernel are protected for AMD64
Address space layout randomization (ASLR)

- Randomize the memory space of a process in order to prevent an attacker from finding addresses or functions
  - Recall our discussions of why some buffer and heap overflow exploits are difficult to deploy in 2016
- Not always present in an OS (e.g., Windows)

http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/
ASLR in Linux

/proc/sys/kernel/randomize_va_space

Mode 2 enabled by default

0 – No randomization. Everything is static.

1 – Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.

2 – Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.