Early Defenses and More Attacks

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 2018
Recap

Vulnerabilities

- Heap overflows can be used to perform arbitrary writes
- Format string vulnerabilities can be used to leak memory and perform arbitrary writes

Defenses

- Stack canaries/cookies can be used to detect stack smashing
- Compilers and libraries (libsafe) can add some basic boundary checking to dangerous functions (strcpy, memcpy, etc.)
- Non-executable data regions prevent code injection
  - Strive for Write-XOR-Execute in programs

Modern attacks: return-to-libc
Topics

Stack overflow defenses
- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses
- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses
Topics

Stack overflow defenses
- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses
- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses
Return-to Attacks

What can I do if I control the return address when I cannot inject code?
Return-to-Attacks

What can I do if I control the return address when I cannot inject code?

Return to an existing function (e.g., a libc function)

<table>
<thead>
<tr>
<th>.text</th>
<th>libc</th>
<th>other lib</th>
<th>other lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application code</td>
<td>C library (defines system call wrappers, memory management routines, and other basic facilities)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```bash
$ ldd /bin/ls
linux-vdso.so.1 (0x00007fffc83b62000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edf1000)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edf83d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edefa9000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9ee016000)
libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
```
Return-to-libc (ret2libc) on 32-bits

Replace return address with the address of an **existing** function

Example: system() executes an a program in a new process
Shell Using ret2libc

Locate system libc call

- `int system(const char *command);`

Set return address to the address of `system()`

```
$ readelf -s /lib/i386-linux-gnu/libc-2.19.so | grep system
1442: 0003de80  56 FUNC  WEAK DEFAULT 12 system@@GLIBC_2.0
```

Prepare one argument for `system()`
int main(void)
{
    system("/bin/shell");
    return 0;
}

080483fb <main>:
080483fb: 8d 4c 24 04
080483ff: 83 e4 f0
08048402: ff 71 fc
08048405: 55
08048406: 89 e5
08048408: 51
08048409: 83 ec 04
0804840c: 83 ec 0c
0804840f: 68 c0 84 04 08
08048414: e8 b7 fe ff ff
Preparing the Stack

Stack

EIP

ESP

804840f: 68 c0 84 04 08  push $0x8048c0
8048414: e8 b7 fe ff ff  call 80482d0 <system@plt>
Preparing the Stack

EIP

804840f: 68 c0 84 04 08 push $0x80484c0
8048414: e8 b7 fe ff ff call 80482d0 <system@plt>

Stack

*cmd

ESP
Preparing the Stack

804840f:  68 c0 84 04 08  push $0x80484c0
8048414:  e8 b7 fe ff ff  call  80482d0 <system@plt>

ESP

EIP  0003de80 __libc_system>:
   3de80:  53  push %ebx

Stack

ret  *cmd

ESP
Preparing the Stack

The stack needs to look like this when `system()` is entered

```
ESP
0003de80 <__libc_system>:
3de80:       53                      push   %ebx
```

```
EIP 0003de80 <__libc_system>:
```

```
804840f:       68 c0 84 04 08          push   $0x80484c0
8048414:       e8 b7 fe ff ff          call   80482d0 <system@plt>
```

Stack:
- `ret`
- `*cmd`
Preparing the Stack

Add a fake return address and a pointer to the command we want to execute on the stack
Return-to-libc on 64-bits

Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
  - RBP, RBX, and R12–R15 are callee saved
  - RAX used for function return
int main(void)
{
    system("/bin/shell");
    return 0;
}

How to load an argument to a register (e.g., rdi)?

00000000000400506 <main>:
  400506:   55                      push %rbp
  400507:   48 89 e5               mov %rsp,%rbp
  40050a:   bf a4 05 40 00         mov $0x4005a4,%edi
  40050f:   e8 cc fe ff ff         callq 4003e0 <system@plt>
Code-reuse Attacks

Any code that already exists in the process can be executed

For example, the following sequence

0x000000000000405255 : pop rdi ; ret

Such short instructions sequences are referred to as gadgets
Return-to-libc on 64-bit

Redirect control to gadget

g1 : pop rdi

Stack

RSP

g1

g1+1 : ret
Return-to-libc on 64-bit

Redirect control to gadget
Load argument on register

Stack

RIP → g1 : pop rdi
       g1+1 : ret

RSP
Redirect control to gadget
Load argument on register
Redirect control to libc function

Stack

RIP

RSP

g1 : pop rdi
f1 <__libc_system>:
g1+1 : ret
f1 : push rbp
Return-to-libc on 64-bit

Redirect control to gadget
Load argument on register
Redirect control to libc function

Stack:

\[
\begin{array}{c|c|c|c|}
 & g1 & *cmd & f1 \\
\hline
RIP & & & \\
\hline
RSP & & & \\
\end{array}
\]

\[
\begin{align*}
g1 &: \text{ pop rdi} \\
g1+1 &: \text{ ret} \\
f1 &: \text{ push rbp}
\end{align*}
\]
Return-to-libc on 64-bit

Redirect control to gadget
Load argument on register
Redirect control to libc function
Get shell!!

Stack

g1  *cmd  f1

RIP

f1 <__libc_system>:
f1 : push rbp
g1 : pop rdi
g1+1 : ret

RSP
Topics

Stack overflow defenses
- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses
- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses
ASCII Armored Address Space

Stack

RSP

g1 *cmd f1

Stack

ESP

ret (libc func) fake ret arg1 arg2
ASCII Armored Address Space

Attacker needs to inject an address and then some

Stack

Stack

ESP

RSP

g1

*cmd

f1

ret

ret (libc func)
ake ret

arg1

arg2
Observation: `strcpy()` stops copying on the first null byte! 

Load libraries in addresses where the first byte is 0x00 (0x00xxxxxx)
ASCII Armored Address Space

Stack

RSP

Cannot overwrite enough bytes

ESP

g1  *cmd  f1

Stack

ret (libc func)  fake ret  arg1  arg2
Problems

Other methods of copying data may not have the same limitation: `memcpy()`, `gets()`, `read()`, `fread()`, custom copy routines, etc.
Topics

Stack overflow defenses
- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses
- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses
Fixed Process Layout

Layout is fixed across all instances of a specific system version → ret2libc attack are possible

$ ldd /bin/ls
    linux-vdso.so.1 (0x0000000000000000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x0000000000000000)
    libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x0000000000000000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000000000000000)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x0000000000000000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x0000000000000000)
    /lib64/ld-linux-x86-64.so.2 (0x0000000000000000)
    libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x0000000000000000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x0000000000000000)
One Attack Fits All

Layout is fixed across all instances of a specific system version → ret2libc attack are possible

An exploit developed on one system will work on all other systems running the same software

$ ldd /bin/ls

linux-vdso.so.1 (0x00007fffc83b62000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edf6df1000)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edf8be8000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9ee0016000)
libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
Disrupt exploits by:

- Randomly choosing the base address of stack, heap, and code segments
- Randomize location of Global Offset Table
- Contains pointers to all functions/globals exported by a library
Base addresses are randomly selected from within predetermined ranges.
Libraries are loaded in the gaps.
Example

```c
unsigned long getEBP (void) {
    __asm ( "movl %ebp ,%eax " );
}

int main(void) {
    printf("EBP: %x\n", getEBP());
}
```

No ASLR

```
> ./getEBP
EBP:bffffff3b8
> ./getEBP
EBP:bffffff3b8
```

With ASLR

```
> ./getEBP
EBP:bf9114c8
> ./getEBP
EBP:bf9114c8
```
ASLR in Linux

First implementation from the PaX project

- [https://pax.grsecurity.net/](https://pax.grsecurity.net/)

Now part of the vanilla kernel
ASLR in Linux

Rs: number of bits randomized in the stack area
Rm: number of bits randomized in the mmap() area
Rx: number of bits randomized in the main executable area
Ls: least significant randomized bit position in the stack area
Lm: least significant randomized bit position in the mmap() area
Lx: least significant randomized bit position in the main executable area

32-bit Linux
Rs = 24, Rm = 16, Rx = 16, Ls = 4, Lm = 12, Lx = 12

64-bit Linux
Much larger entropy
ASLR in Windows

Vista and Server 2008

Stack randomization
- Find Nth hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)

Heap randomization: 5 bits
- Linear search for base + random 64K-aligned offset

EXE randomization: 8 bits
- Preferred base + random 64K-aligned offset

DLL randomization: 8 bits
- Random offset in DLL area; random loading order
Weak Randomization

Weak random number generators, implementation bugs, etc.
Biased Selection of Heap Base Address

“An Analysis of Address Space Layout Randomization on Windows Vista”, Ollie Whitehouse, BlackHat 2007

Brute-forcing ASLR

Exploiting server software using fork()

Disk

Server program

library

Memory

Program image

library

library

library

Randomize location of program and libraries in memory

Loader
Brute-forcing ASLR

Exploiting server software using fork()

Memory

Program image
library
library
library

Process 1
fork()

Process 2

Child process shares layout with parent

Incoming user request
Brute-forcing ASLR

Exploiting server software using fork()
Brute-forcing ASLR

Exploiting server software using fork()

Memory

Program image
library
library
library

Process 1
fork()
Process 2

Incoming user request

Attack child process

Repeat till successful
For ASLR to be applied to code it needs to be position independent

Libraries ➔ Position Independent Code (PIC)

Executables ➔ Position Independent (PIE)
Exploit the Weakest Link

A single non-randomized library may be enough
Exploit the Weakest Link

Do not forget the program image

Program image
library
library
library
Program image
library
library
library
Program image
library
library
library
Exploit the Weakest Link

Executables became PIE recently

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Tested Binaries</th>
<th>PIE Enabled</th>
<th>Not PIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu 12.10</td>
<td>646</td>
<td>111 (17.18%)</td>
<td>535</td>
</tr>
<tr>
<td>Debian 6</td>
<td>592</td>
<td>61 (10.30%)</td>
<td>531</td>
</tr>
<tr>
<td>CentOS 6.3</td>
<td>1340</td>
<td>217 (16.19%)</td>
<td>1123</td>
</tr>
</tbody>
</table>

Percentage of PIE binaries in different Linux distributions
PLT entry consists of 3 instructions
  - First jumps to address contained in the GOT
  - Initially pointing to the linker → will resolve the function and update the GOT

Functions are bound lazily → on first call
An information leak is caused by exploiting a bug that discloses the memory layout and/or contents of a program.

Main idea:
- Corrupting (partially) data that affect what or how much is read from memory
- Receive the output of the read
Leak Can Occur in the Stack

```c
void func(char *filename, int len)
{
    char path[128] = "/tmp/";
    memcpy(path, filename, len);
    ...
    fprintf(logfl, "Opened %s\n", path);
    ...
}
```

Omitting or overwriting the terminating ‘\0’ character and reading a string can leak data
Or the Heap

```cpp
void string::copy(string *src)
{
    ...
    memcpy(this->data, src->data, src->len);
    ...
}

outputfile->copy(userinput);
...
logfl << "user entered " << userinput << endl;
```
Or the Heap

```cpp
void string::copy(string *src)
{
    ...
    memcpy(this->data, src->data, src->len);
    ...
}

outputfile->copy(userinput);
...
logfl << "user entered " << userinput << endl;
```

```cpp
class string
{
    ...
    private:
        size_t len;
        char *data;
    ...
};
```
Memory leaks
  - Combine memory leaks with control-flow hijacking
  - Repeatable arbitrary memory leaks are better

Insufficient entropy

Incompatible binaries

...
Attacks in the Information Leak Era

Many of the other bugs we have already seen can be used to leak information

- Overflow
- Use-after-free
- Type confusion

JavaScript is frequently used as it allows dynamically triggering the exploit multiple times to

- Leak data
- Hijack control flow