Early Defenses and More Attacks

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 2018
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
Insert special values, called canaries, between local variables and function return address

Canary values are inserted on function entry

Canaries are verified before a function returns
- Program stops if the canary has changed
Stack Overflow With Canary

```c
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}
```

./mytest AAAAA

- High address/stack bottom
- RETADDR
- canary
- buf
- buf
- buf
- buf
- buf
- Low address/stack top
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAAAAAAAAA
Canary Types

Random canary: (used in Visual Studio, gcc, etc.)
  ▪ Choose random string at program startup
  ▪ Insert canary string into every stack frame
  ▪ Verify canary before returning from function
  ▪ To corrupt random canary, attacker must learn current random string

Terminator canary:
  Canary = 0 (null), newline, linefeed, EOF
  ▪ String functions will not copy beyond terminator
  ▪ Hence, attacker cannot use string functions to corrupt stack.
Example: C code

```c
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("len: %ld\n", strlen(buf));
    return strlen(buf);
}
```
### Example: Compiled Code

```
0000000000400606 <mytest>:
  400606:  55                      push %rbp
  400607:  48 89 e5                mov %rsp,%rbp
  40060a:  48 83 ec 30              sub $0x30,%rsp
  40060e:  48 89 7d d8              mov %rdi,-0x28(%rbp)
  400612:  64 48 8b 04 25 28 00    mov %fs:0x28,%rax
  400619:  00 00                    mov %rax,-0x8(%rbp)
  ...                   
  40065e:  48 8b 4d f8              mov -0x8(%rbp),%rcx
  400662:  64 48 33 0c 25 28 00    xor %fs:0x28,%rcx
  400669:  00 00                    mov -0x8(%rbp),%rcx
  40066d:  74 05                    xor %fs:0x28,%rcx
  400672:  c9                      callq 4004d0 <__stack_chk_fail@plt>
  400673:  c3                      leaveq
```

*Store canary*

*Verify canary*
Alignment of Stack Buffers and Canaries

The order of local variables may be important
Alignment of Stack Buffers and Canaries

The order of local variables may be important

Buffer overflows could allow important local variables to be controlled
Alignment of Stack Buffers and Canaries

Place canary between buffer and saved ebp/return address

The compiler may not always be able to align stack variables “ideally”
StackShield

Address obfuscation instead of canary

Encrypt return address on stack by XORing with random string

Decrypt just before returning from function

Attacker needs decryption key to set return address to desired value.
Example: StackShield

```c
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s
", buf);
    return strlen(buf);
}
```
Example: StackShield

int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}
Problems

Canaries can be omitted in small functions or non-string buffers

Canaries/keys can be leaked

Bugs may leave canaries untouched
Problems

From GCC’s documentation

-fstack-protector
 Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions **with buffers larger than 8 bytes**. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits

Can be disabled with **-fno-stack-protector** flag
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
Run time checking: Libsafe

Dynamically loaded library
Intercepts calls to `strcpy (dest, src)`
- Validates sufficient space in current stack frame:
  \[|\text{frame-pointer} - \text{dest}| > \text{strlen(src)}\]
- If so, does `strcpy()`
- Otherwise, terminates application
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
Heap Protections

Heap Arbitrary Writes

n->next->prev = n->prev;
n->prev->next = n->next;

Facts About DLinked Lists

n->prev->next == n
n->next->prev == n

If these are violated a corruption has occurred!
Heap Protections

Heap Arbitrary Writes

n->next->prev = n->prev;
n->prev->next = n->next;

Facts About DLinked Lists

n->prev->next == n
n->next->prev == n

If these are violated a corruption has occurred!
Other Protections

Separating metadata from chunks

Adding canary type values
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
Virtual Memory

Virtual memory

Physical memory
The Memory Management Unit

Used in all modern servers, laptops, and smart phones

One of the great ideas in computer science
For many years the read permission implied execute as well.
Non-executable Memory (PaX)

PaX stands for PageEXec

Introduced in 2000

A Linux kernel patch protection emulating non-executable memory

PaX refused code execution on writable pages
Each page is associated with a supervisor bit
  - Access only allowed from the kernel

PaX set that bit and kept track of PaX-protected pages

Page-fault handler intercepted to check for PaX-protected pages
Each page is associated with a supervisor bit

- Access only allowed from the kernel

PaX set that bit and kept track of PaX-protected pages

Page-fault handler intercepted to check for PaX-protected pages
Processor manufacturers introduced a new bit in page permissions to prevent code injections.

Coined **No-eXecute** or **Execute Never**

The NX-bit (No-eXecute) was introduced first by AMD to resolve such issues in 2001.

- Asserting NX, makes a readable page non-executable
- Frequently referred to as Data Execution Prevention (DEP) on Windows

**Marketed as antivirus technology**
Enhanced virus protection

Costin Raiu  Kaspersky Lab

download slides  (PDF)

AMD Athlon 64 CPU Feature:

1. HyperTransport technology
2. Cool'n'Quiet technology
3. Enhanced Virus Protection for Microsoft Windows XP SP2

The AMD64 architecture is an affordable way of getting the power of 64-bit processing into a desktop computer. Interesting enough, AMD has not only designed an improved CPU core and longer registers, but they have also included a feature designed to significantly increase the security of modern operating systems.

The idea of hardware protection isn't new – every contemporary CPU includes at least a basic hardware mechanism for enforcing a security scheme, for instance, those from the Intel x86 family, based on the NX bit. The main difference is that this feature is very widespread and has a great potential to detect a malware attack, even in virtual environments.
Adoption

A non-executable stack was not immediately adopted

The OS occasionally needed to place code in the stack

- For example, trampoline code for handling UNIX signals
The Write XOR Execute (W^X) policy mandates that in a program there are no memory pages that are both writable and executable
Malicious code (shellcode) is injected into attacker controlled, executable memory.

The controlled instruction pointer is directed to injected code.
Unless You Are a Browser...

Very popular software
  - Probably installed on every client device

Large and complex software

Execute JavaScript
How Does JavaScript Run

1. Source code
2. Parser
3. AST
4. Bytecode generator
5. Bytecode
6. Interpreter

Execution
How Does JavaScript Run

Source code → Parser → AST → Bytecode generator → Bytecode → JIT compiler → Native code → JITed code → Code Cache → Execution

 execution
How Does JavaScript Run

- Google V8 designed specifically to execute at speed.
- Bytecode generation skipped
- Directly emit native code
- Overall JavaScript execution improved by 150%
JITed code and code cache have interesting properties from the perspective of the attacker

- Code is continuously generated
- Code needs to be executable

Violates the W^X policy
JITed code and code cache have interesting properties from the perspective of the attacker

- Code is continuously generated
- Code needs to be executable

Violates the W^X policy
From JS to Code Cache

JS code is JITed and placed in the code cache

Some JS engines do not separate data and code

```html
<html>
<body>
<script language='javascript'>
var myvar = unescape('%u4F43u4552'); // CORE
myvar += unescape('%u414C%u214E'); // LAN!
alert("allocation done");
</script>
</body>
</html>
```
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)

Process

<table>
<thead>
<tr>
<th>.text</th>
<th>libc</th>
<th>other lib</th>
<th>other lib</th>
</tr>
</thead>
</table>

Application code

C library (defines system call wrappers, memory management routines, and other basic facilities)
```
$ ldd /bin/ls

linux-vdso.so.1 (0x00007ff83b62000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edf90f100)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edf90f100)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edfb57000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edfb55000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf90f100)
/lib64/ld-linux-x86-64.so.2 (0x00007f9edf90f100)
libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf90f100)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edf90f100)
```
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

- \textit{int system(const char *command);}\

Set return address to the address of \textit{system()}

```sh
$ 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 \textit{system()}
int main(void) {
    system("/bin/shell");
    return 0;
}

080483fb <main>:
    8d 4c 24 04          lea  0x4(%esp),%ecx
    83 e4 f0                and $0xffffffff0,%esp
    ff 71 fc        pushl  -0x4(%ecx)
    55                      push   %ebp
    89 e5                      mov   %esp,%ebp
    51                      push   %ecx
    83 ec 04          sub    $0x4,%esp
    83 ec 0c          sub    $0xc,%esp
    68 c0 84 04 08        push $0x80484c0
    e8 b7 fe ff ff    call  80482d0 <system@plt>
    ...
Preparing the Stack

\[
\begin{align*}
804840f: & \quad 68 \text{ c0 84 04 08} \quad \text{push} \quad 0x80484c0 \\
8048414: & \quad e8 \text{ b7 fe ff ff} \quad \text{call} \quad 0x80482d0 \quad \text{<system@plt>}
\end{align*}
\]
Preparing the Stack

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

Stack

```
ESP
*cmd
```
Preparing the Stack

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

EIP: 0003de80 __libc_system:  
3de80:  53

```
Stack
```

- **ESP**
- **EIP**
- **ret**
- **cmd**
Preparing the Stack

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

Stack

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

ESP

EIP

The stack needs to look like this when `system()` is entered
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
```c
int main(void)
{
    system("/bin/shell");
    return 0;
}
```

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

```
0000000000400506 <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

\[
g1 : \text{pop rdi} \\
g1+1 : \text{ret}
\]
Return-to-libc on 64-bit

Redirect control to gadget
Load argument on register

Stack

RIP

RSP

g1 : pop rdi
g1+1 : ret

*cmd
Return-to-libc on 64-bit

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

Stack

g1
*cmd
f1

RIP

g1  : pop rdi
g1+1 : ret

f1 <__libc_system>:
f1 : push rbp
Return-to-libc on 64-bit

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

g1 : pop rdi
   g1+1 : ret

f1 <__libc_system>: 
   f1 : push rbp

Stack
   g1  
   *cmd
   f1

RIP
RSP
Return-to-libc on 64-bit

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

Stack

\[
\begin{align*}
g1 & : \text{pop rdi} \\
g1+1 & : \text{ret} \\
f1 & <\_\_\_\_\_\text{libc_system}>: \\
RIP & \rightarrow f1 : \text{push rbp}
\end{align*}
\]