Recap: Broadly Deployed Security Mechanisms

NX-bit → Prevent arbitrary code execution

Stack canaries → Detect and prevent stack overflows

ASLR → Introduce uncertainty on the location of injected shellcode and existing code in a running program

They have raised the bar for attackers
Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP
Control-flow Integrity (CFI)
Attacks against CFI and more defenses
Topics

Attacks shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP
Control-flow Integrity (CFI)
Attacks against CFI and more defenses
Shift in Target Selection

Clients

Servers
Shift in Target Selection

Clients

- Web browsers
  - Firefox
  - Safari
  - Chrome

- Flash
- Acrobat Reader

Servers
Shift in Target Selection

**Clients**

- Web browsers
- Flash
- Acrobat Reader

**Why?**

1. **Software popularity**
2. **Large and complex software**
   - More buggy
3. **Dynamically translates and executes Javascript**
   - Attackers can run code on target (even if in isolation)
Recap: Code Injection in the Code Cache

```html
<html>
<body>
<script language='javascript'>
var myvar = unescape('%u4F43%u4552'); // CORE
myvar += unescape('%u414C%u214E'); // LAN!
alert("allocation done");
</script>
</body>
</html>
```

ASLR → Code cache location **unknown**
Heap Spraying

Attempt to place shellcode at a predictable location

Mechanisms:
Dynamically expand buffer by appending copies of the shellcode
On the fly generate variables

var v1 = “myshellcode”;
var v2 = “myshellcode”;
var v3 = “myshellcode”;
```javascript
var v1 = "myshellcode";
var v2 = "myshellcode";
var v3 = "myshellcode";
var v4 = "myshellcode";
```
Large NOP Sleds
Spray up to your predictable location

Consecutive chunks of nops + shellcode
Summary: Heap Spraying

May require multiple attempts

Can possibly defeat ASLR

Heap fragmentation is in play
  - May be worse in concurrent systems
Code/Data Separation in the Code Cache

- JIT compiler
- Native code
- Static data
- Code Cache
- Execution
- Bounded code cache size
- Dynamically allocated data
- Heap
ASLR + Code/data Separation + Finite Code Cache

No More Code Injection
Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses
Chaining Functions with ret2libc
Chaining Functions with ret2libc

F1(cmd)    F2(arg1, arg2)

Stack

ESP

fakeret    *cmd
Chaining Functions with ret2libc

F1(cmd)  F2(arg1, arg2)

Stack
Chaining Functions with ret2libc

Stack

F1(cmd)  F2(arg1, arg2)

ESP
Chaining Functions with ret2libc

F1(cmd)  F2(arg1, arg2)  F3(arg3)

Stack

F1(cmd)  F2(arg1, arg2)  F3(arg3)

Stack

*cmd  arg2  arg1

ESP
Chaining Functions with ret2libc

F1(cmd)  F2(arg1, arg2)  F3(arg3)

Stack

*cmd  arg2  arg1
Chaining Functions with ret2libc

We need small gadgets to unwind the stack pointer in a controlled way
Chaining Functions with ret2libc

F1(cmd)
Chaining Functions with ret2libc

F1(cmd)
pop eax; ret

![Diagram of stack with ESP, *cmd, F2, ret, arg2, arg1]
Chaining Functions with ret2libc

F1(cmd)
pop eax; ret
F2(arg1, arg2)
Chaining Functions with ret2libc

F1(cmd)
pop eax; ret
F2(arg1, arg2)
add 0x8, esp; ret
Chaining Functions with ret2libc

F1(cmd)
pop eax; ret
F3(arg1, arg2)
add 0x8,esp; ret
Chaining Functions with ret2libc

F1(cmd)
pop eax; ret
F2(arg1, arg2)
add 0x8, esp; ret
F3(arg3)
0x0804851c <+88>: leave //mov ebp, esp; pop ebp;
0x0804851d <+89>: ret     //return
Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP
Control-flow Integrity (CFI)
Attacks against CFI and more defenses
I don’t like only calling functions
Enter Return-Oriented Programming

Re-use parts of the application’s code (gadget) to perform arbitrary computations

A Turing complete machine

Use the stack like a tape providing the data for the computation and the instruction pointer
A Code Collage
mov (%rcx),%rbx
test %rbx,%rbx
je 41c523 <main+0x803>
mov %rbx,%rdi
callq 42ab00
mov %rax,0x2cda8d(%rip)
cmpb $0x2d,(%rbx)
je 41c4ac <main+0x78c>
mov 0x2cda8d(%rip),%rax
ret
test %rbx,%rbx
mov $0x4ab054,%eax
cmove %rax,%rbx
test %rdi,%rdi
je 41c0c2 <main+0x78c>
mov %rbx,0x2cda6a(%rip)
test %rdi,%rdi
je 41c0c2 <main+0x78c>
mov %rax,0x2d2945(%rip)
mov 0x2cda16(%rip),%rax
test %rax,%rax
je 41c112 <main+0x3f2>
movzbl (%rax),%edx
cmp $0x2d,%r12b
je 41c440 <main+0x720>
xor %ebp,%ebp
mov $0x4c223a,%ebx
add $0x1,%r14
jmp 41c1a3 <main+0x483>
cmp (%rbx),%r12b
mov %ebp,%r13d
jne 41c188 <main+0x468>
mov %rax,0x2d2670(%rip)
test %eax,%eax
xchg %ax,%ax
je 41c188 <main+0x468>
movslq %r15d,%rax
mov (%rdx,%rax,8),%r14
ret
je 41c214 <main+0x4f4>
cmp $0x1,%r15d
movslq %ebp,%rax
ret
jne 41c214 <main+0x4f4>
cmpb $0x2d,(%r14)
jne 41c188 <main+0x468>
movzbl 0x1(%r14),%r12d
movl $0x0,0x18(%rsp)
cmp $0x2d,%r12b
je 41cef6 <main+0x11dd>
movslq %r15d,%rax
mov (%rdx,%rax,8),%r14
ret
je 41c214 <main+0x4f4>
cmpb $0x2d,(%r14)
jne 41c188 <main+0x468>
movzbl 0x1(%r14),%r12d
movl $0x0,0x18(%rsp)
cmp $0x2d,%r12b
je 41cefd <main+0x11dd>

Gadgets
An Example

<table>
<thead>
<tr>
<th>Payload</th>
<th>Code</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0xb8800000: pop eax ret</td>
<td></td>
</tr>
<tr>
<td>esp</td>
<td>...</td>
<td>eax = 1</td>
</tr>
<tr>
<td>0xb8800000</td>
<td>0xb8800010: pop ebx ret</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td>0xb8800020: add ebx, eax ret</td>
<td></td>
</tr>
<tr>
<td>0xb8800010</td>
<td>0xb8800030: mov eax, [ebx] ret</td>
<td></td>
</tr>
<tr>
<td>0xb8800020</td>
<td></td>
<td>ebx = 0x400000</td>
</tr>
<tr>
<td>0xb8800030</td>
<td></td>
<td>*ebx = eax</td>
</tr>
<tr>
<td>0xff</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0xff
Current State of the Art

First-stage ROP code for bypassing NX
- Allocate/set W+X memory (VirtualAlloc, VirtualProtect, ...)
- Copy embedded shellcode into the newly allocated area

Second stage jumps to injected code

Pure-ROP exploits
- In-the-wild exploit against Adobe Reader XI
- CVE-2013-0640
Attacks against CFI and more defenses
Fine-Grained Code Randomization

Randomize the layout of the code within a library/executable

Aims to defeat ROP-style attacks that rely on a memory leak to de-randomize the base address of a code segment

- This allows using the gadgets within

Can be applied at different levels with increasing overheads

- Function
- Basic block
- Instruction
Leak library base address
Known library base address

The address of every instruction is known
Function-level Randomization

Order of functions is randomly selected at compile time

library

function3

function1

function2

library

function1

function2

function3
Basic Block-level Randomization

library

function1

function3

function

BBL4

BBL3

BBL2

BBL1
Basic Block-level Randomization

Order of basic blocks is randomly selected at compile time

Glue code may be inserted
Instruction-level Randomization

Similar concept to function and BBL-level randomization

Instruction may be

- Moved within a block (e.g., by adding random number of NOPs between them)
- Replaced with equivalent functionality
- Substituted to use different registers
- ....
Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)
Attacks against CFI and more defenses
JIT-ROP

Just-In-Time ROP chain generation
Can bypass fine-grained randomization
- When a memory leak can be repeatedly triggered
- Example: Leaks that can be triggered from JS

Main idea:
Dynamically leak memory and locate gadgets for ROP
Construct ROP chain and exploit control-flow hijacking vulnerability

Leak address of a single page
Search for pointers to other pages
Repeat process for newly discovered pages
Just-in-time Disassembly

Disassemble pages and scan for useful gadgets

page
page
page
page
Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP

Control-flow Integrity (CFI)
Attacks against CFI and more defenses
Attacker Modus Operandi

Find memory corruption bug
- Manipulate to take over program counter

Find ASLR bypass
- Leak memory layout
- Spray memory
- Weakly or non-randomized sections/memory

Inject ROP payload
- Break W^X semantics

Inject code
Attacker Modus Operandi

Find memory corruption bug

- Manipulate to take over program counter

Control-flow Integrity aims to restrict the arbitrary manipulation of the program counter
Control Flow Manipulation

Function calls

```c
my_function(arg1, arg2)

void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

Function returns

```c
return;
return 100;
```

If statements

```c
if (cond) {
} else {
}
```

Loops

```c
for () {} while {} do {} while
```

Break/continue

```c
while (true) {
    if (cond)
        break;
}
while (cond) {
    if (cond2)
        continue;
}
```

Switch statement

```c
switch (cond) {
    val1: ... break;
    val2: ... break;
}
```

goto statement

```c
goto label1;
... Label1:
```
Control-Flow Hijacking Prone Statements

Statements where the target statement cannot be known a priori

- Indirect control-flow transfers

Indirect calls, returns, and some switches

Calls to virtual functions are indirect calls

```c
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);

Class C {
    virtual void vcall(void);
}
C obj = new C();
obj->vcall();
```
Easily Observable in Machine Code

**C Code**

```c
void (*fptr)(arg1_type, arg2_type) = &my_function;

fptr(arg1, arg2);

return;

return 100;

switch (cond) {
    val1: ... break;
    val2: ... break;
}

Class C {
    virtual void vcall(void);
}

C obj = new C();

obj->vcall();
```

**Machine Code**

```
ret

ejmp *(%rax)

ejmp *(%rax)

call *(%rax)

call *(%rax)
```
Function Call Graph (FCG)
FCG Enforcement

parent_function() → call → next_function() → call → function()
Control-flow Graph (CFG)

Indirect flows only

parent_function()

function()

next_function():

jump *

call
CFI - CFG Enforcement

parent_function():

next_function():

function():

ret

call

call

call

call *

jmp *
Extracting the CFG

With source code
- More reliable
- Still not perfect
- How to handle
  - Dynamically loaded libraries?
  - Callbacks

Without source code
- Requires accurate disassembly
- Cannot accurately define all paths
- Shared libraries are easier to handle

```c
static void (*fptr)(char *string, int len);
void set_callback(void *ptr)
{
    fptr = ptr;
}
void process_items()
{
    for (string *s : items) {
        fptr(s->c_str, s->len);
    }
}
```
Working with an Imperfect CFG

Lets assume that we know/can learn

- The location of every function
- The location of every indirect branch instruction

Coarse-grained CFI can enforce the following

- Indirect calls should only transfer control to functions
  - Same for most jumps

- Returns should only transfer control to instructions following a indirect call or jump
call *(%rax)
call *(%rax) → Function_A: ret

Function_B: ret

OK

OK
Function_A:

ret

Function_B:

call *(%rax)

OK

pop %rax

ret

NOT

OK