Memory safety, continued

With material from Mike Hicks, Dave Levin and Michelle Mazurek

Today

- Return Oriented Programming
  - Yet another type of buffer overflow attack
    - Bypasses countermeasures discussed last time
- Control Flow Integrity
  - General countermeasure against buffer overflow attack
    - Can detect if logical flow of program is interrupted
- Other types of overflow attacks
Return oriented programming (ROP)
Return-oriented Programming

• Introduced by Hovav Shacham, CCS 2007

• Idea: rather than use a single (libc) function to run your shellcode, *string together pieces of existing code, called gadgets*, to do it instead

• Challenges
  • Find the gadgets you need
  • String them together
Approach

- Gadgets are instruction groups that end with `ret`.
- Stack serves as the code.
  - `%esp = program counter`.
- Gadgets invoked via `ret` instruction.
- Gadgets get their arguments via `pop`, etc.
  - Also on the stack.
Simple example

Goal: put 5 into edx

mov %edx, 5

"program counter"

"Instructions"
Code sequence (no ROP)

0x17f: mov %eax, [%esp]
mov %ebx, [%esp+8]
mov [%ebx], %eax

%eip
%esp
%eax  5
%ebx  0x404

0x00  0x404  0xffffffff

Text  5  ...  5  ...  0x404  ...  ...
Equivalent ROP sequence
Return-Oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.
Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - Automated search: look for `ret` instructions, work backwards
    - Cf. https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?
  - For significant codebases (e.g., libc), Turing complete
    - Especially true on x86’s dense instruction set
  - Schwartz et al. (USENIX Sec’11) automated gadget shellcode creation, Turing complete not required
Control Flow Integrity
Behavior-based detection

- Stack canaries, non-executable data, ASLR make standard attacks harder / more complicated, but may not stop them

- Idea: **observe** the program’s **behavior** — **is it doing what we expect it to?**
  - If not, might be compromised

- Challenges
  - Define “expected behavior”
  - Detect deviations from expectation efficiently
  - Avoid compromise of the detector
Control-flow Integrity (CFI)

- Define “expected behavior”:
  - Control flow graph (CFG)
  - Detect deviations from expectation efficiently

- Avoid compromise of the detector

Reference:
Which functions call other functions

```cpp
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```
Control Flow Graph

```c
bool lt(int x, int y) {
    return x<y;
}

bool gt(int x, int y) {
    return x>y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```

*Break into basic blocks*
*Distinguish calls from returns*
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
  - During compilation, or from the binary

- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG

- Observation: **Direct calls** need not be monitored
  - Assuming the code is immutable, the target address cannot be changed

- Therefore: **monitor only indirect calls**
  - `jmp`, `call`, `ret` with non-constant targets
sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}

Control Flow Graph

Direct calls (always the same target)
Control Flow Graph

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}

*Indirect transfer* (call via register, or ret)
Control-flow Integrity (CFI)

- Define “expected behavior”:
  Control flow graph (CFG)

- Detect deviations from expectation efficiently
  In-line reference monitor (IRM)

- Avoid compromise of the detector
In-line Monitor

• Implement the monitor in-line, as a program transformation

• Insert a label just before the target address of an indirect transfer

• Insert code to check the label of the target at each indirect transfer
  • Abort if the label does not match

• The labels are determined by the CFG
Simplest labeling

Use the same label at all targets: label just means it’s OK to jump here.

What could go wrong?
Simplest labeling

- Can’t return to functions that aren’t in the graph
- Can return to the right function in the wrong order
Detailed labeling

- All potential destinations of **same source** must match
  - Return sites from calls to `sort` must share a label `(L)`
  - Call targets `gt` and `lt` must share a label `(M)`
  - Remaining label unconstrained `(N)`

*Prevents more abuse than simple labels,*
*but still permits call from site A to return to site B*
Classic CFI instrumentation

Before CFI

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 53 08</td>
<td>call [ebx+8]</td>
<td>; call a function pointer</td>
</tr>
</tbody>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

| 8B 43 08        | mov eax, [ebx+8]  | ; load pointer into register |
| 3E 81 78 04 78 56 34 12 | cmp [eax+4], 12345678h | ; compare opcodes at destination |
| 75 13           | jne error_label  | ; if not ID value, then fail |
| FF D0           | call eax         | ; call function pointer      |
| 3E 0F 18 05 DD CC BB AA | prefetchnta [AABBCCDDh] | ; label ID, used upon the return |

Fig. 4. Our CFI implementation of a call through a function pointer.

After CFI

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h</td>
<td>; return, and pop 16 extra bytes</td>
</tr>
</tbody>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

| 8B 0C 24        | mov ecx, [esp]    | ; load address into register |
| 83 C4 14        | add esp, 14h      | ; pop 20 bytes off the stack |
| 3E 81 79 04 DD CC BB AA | cmp [ecx+4], AABBCCDDh | ; compare opcodes at destination |
| 75 13           | jne error_label  | ; if not ID value, then fail |
| FF E1           | jmp ecx           | ; jump to return address     |
Classic CFI instrumentation

Fig. 4. Our CFI implementation of a call through a function pointer.

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h</td>
<td>; return, and pop 16 extra bytes</td>
</tr>
</tbody>
</table>

is instrumented using `prefetchnta` destination IDs, to become:

| 8B 0C 24        | mov ecx, [esp]    | ; load address into register |
| 83 C4 14        | add esp, 14h      | ; pop 20 bytes off the stack |
| 3E 81 79 04 DD CC BB AA | cmp [ecx+4], AABBCDDh | ; compare opcodes at destination |
| 75 13           | jne error_label   | ; if not ID value, then fail |
| FF E1           | jmp ecx           | ; jump to return address |
Efficient?

- **Classic CFI** (2005) imposes 16% **overhead** on average, 45% in the **worst case**
  - Works on arbitrary executables
  - Not modular (no dynamically linked libraries)

- **Modular CFI** (2014) imposes 5% **overhead** on average, 12% in the **worst case**
  - C only
  - Modular, with separate compilation
  - [http://www.cse.lehigh.edu/~gtan/projects/upro/](http://www.cse.lehigh.edu/~gtan/projects/upro/)
Control-flow Integrity (CFI)

- Define “expected behavior”:
  
  **Control flow graph** (CFG)

- Detect deviations from expectation efficiently

  **In-line reference monitor** (IRM)

- Avoid compromise of the detector

  **Sufficient randomness, immutability**
Can we defeat CFI?

- **Inject code** that has a legal label
  - *Won’t work* because we assume non-executable data

- **Modify code labels** to allow the desired control flow
  - *Won’t work* because the code is immutable

- **Modify stack during a check**, to make it seem to succeed
  - *Won’t work* because adversary cannot change registers into which we load relevant data
CFI Assurances

• CFI defeats control flow-modifying attacks
  • Remote code injection, ROP/return-to-libc, etc.

• But not manipulation of control-flow that is allowed by the labels/graph
  • Called mimicry attacks
  • The simple, single-label CFG is susceptible to these

• Nor data leaks or corruptions
  • Heartbleed would not be prevented
  • Nor the authenticated overflow
    • Which is allowed by the graph

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, str);
    if(authenticated) { ... }
}
Secure?

- MCFI can **eliminate 95.75% of ROP gadgets** on x86-64 versions of SPEC2006 benchmark suite
  - By ruling their use non-compliant with the CFG

- **Average Indirect-target Reduction (AIR) > 99%**
  - Essentially, the percentage of possible targets of **indirect jumps** that CFI rules out