Instruction Set Filters and Other Exploit Defenses
Changing the architecture to make exploitation harder.

Taylor Hornby    Michael Locasto

October 29, 2013
All sufficiently complex software contains vulnerabilities.

We want to run vulnerable software and stay safe.

Defenses developed in response to specific attacks/techniques.

Attacks developed in response to defenses.
Background
The not-so-periodic table of attack and defense

Data Manipulation
- Use After Free
- Arbitrary Write
- Heap Overflow
- Stack Overflow
- Memory-safe languages
- Correct Code
- SQLI, XSS, RFI
- Not used
- Human Error (Omission)

Control Flow Manipulation
- Data-Only Attacks
- Return Address Overwrite
- Stack Canaries
- Overwrite Data Pointer
- kBouncer
- ROPGuard
- Jump Oriented Programming (JOP)
- To Program Code (ROP)
- To Program Code (ROP)
- Only return to after a CALL

Guard Pages
- To Data
- To libc

Randomized Allocation
- In Stack
- In Heap
- Brute Force
- To Program Code

Brute Force
- W^X
- Partial Address Overwrite
- Information Disclosure
- There are still useful gadgets.
Attacker has to express their malicious computation somehow.

Think of defenses as limiting the attacker’s ability to express their malicious computation.

1. Prevent attacker from “speaking” the language.
   - $W \oplus X$ (DEP)
   - Stack canaries
   - XFI

2. Make the language unpredictable.
   - ASLR
   - Instruction set randomization

3. Make the language smaller or less powerful.
   - RET always returns to an instruction after a CALL
   - Detect unusual call/jump sequences.
   - Enforce an order in which functions can be executed.
   - Instruction Set Filters
Background
Return Oriented Programming

Motivation

- We can’t inject our own code because of $W \oplus X$.
- We can’t return to `system()` because of ASLR.
- Let’s re-use the application’s code to perform our computation.
- Find useful code snippets (called *gadgets*) that end in RET.
- Stitch them together to perform our computation.
The stack contains our ROP program.
The stack pointer (ESP) is the new program counter.
Background
Return Oriented Programming Defenses

- kBouncer
  - Vasilis Pappas, 2012
  - Winner of Microsoft BlueHat Prize ($200,000)
  - Use Last Branch Recording to keep history of code path.
  - When entering Win32 API call, look for ROP-like patterns.

- Smashing The Gadgets
  - Vasilis Pappas et al., 2012
  - Substitute equiv. instructions (randomizes unintended instrs).
  - Register re-assignment.
  - Randomize the order of instructions.
  - Program does the same thing, but gadgets break.
Need more rigorous way of evaluating defenses.

Borrow from cryptography: Model it as a game.

**Chosen-PC Attack (CPCA)**
1. Attacker receives the process’s memory and registers.
2. Attacker sends a list $L$ of $N$ executable addresses.
3. For each address $L_i$, start executing at $L_i$, then just before the next indirect call or jump, go to $L_{i+1}$.

**Adaptive Chosen-PC Attack (ACPCA)**
1. Attacker receives the process’s memory and registers.
2. Attacker selects an address $A$.
3. Execution starts at $A$ until the next indirect call or jump.
4. Go back to step (1).

These encompass all code reuse attacks.

Even with ACPCA-security, *non-control data* attacks are possible.
Instruction Filters
Overview

- Disable parts of the instruction set based on context.
- Protected shadow stack holding the current filters.
- If an exploit is triggered in parse(), INT won’t be available.
The average number of gadgets that would be allowed by a function’s instruction filter.

1

1Do not trust this data too much
Random sample of gadget sequences of different length.

- Complete: Make call graph complete (jump to any filter)
- All: Can traverse any edge of call graph
- Reverse: Can only traverse call edges backwards (returns).

\[\text{Do not trust this data too much}\]
If you could inject machine code, how much of a shellcode could you execute? Sample: 200 shellcodes from shell-storm.org.

Do not trust this data too much

\[^3\]
We filter on the opcode number, because it’s easy and fast.

Instructions are mapped to integer between 0x000 and 0x3FF

Opcodes are either 1 byte, 2 bytes, or 3 bytes:

1. Opcode = 0x??, Number = 0x0??
2. Opcode = 0x0F??, Number = 0x1??
3. Opcode = 0x0F38??, Number = 0x2??
4. Opcode = 0x0F3A??, Number = 0x3??

---

4 Taken from Intel Software Developer’s Manual Volume 2C
Implementation
Registers and Memory

**Registers**
- FSB (32 bits)
- FST (32 bits)
- FAB (32 bits)
- Current Filter (1024 bits)

**Memory**
- Filter Stack
  - 1
  - 3
  - 2
  - 7
  - (Top of stack contains current filter ID)
  - (Grows down as functions are called)
- Filter List (ELF Segment)
  - Filter 0
  - Filter 1
  - Filter 2
  - ...
Added Instructions
- FLOW $n$: Push $n$ onto the filter stack.
- UNFLOW $n$: Pop $n$ from the filter stack.
- FCHECK $n$: Assert current filter ID is $n$.
- Privileged instructions for setup and context switching.

Memory between FST and FSB can only be modified by FLOW or UNFLOW.

Using PEBIL to add filters to ELF binaries.
- http://www.sdsc.edu/PMaC/projects/pebil.html
My implementation doesn’t really work.

This is my fault, not because the idea is bad.

Problem is with the static instrumentation.
  
  Lots of crazy code that needs manual filter exceptions.
  
  PEBIL changes the code in weird ways.

So I just disable the filter for everything that doesn’t work, which eliminates many of the security properties.

The right place to do this is in the compiler.

But I can show you it stopping an attack...
Demo
Breaks existing exploits with high probability.

What about exploits designed with knowledge of the defense?

Attacker can:

- Execute code allowed by the current filter.
- Execute UNFLOW $n$, which pops the current filter off the stack and enables the previous one (unless the stack is empty).
- Execute FLOW $n$, which switches to a different filter. These are always at the start of procedures.

If the attacker wants to perform some computation, they have to search for a sequence of filters that will let it execute, then find a way to switch into those filters \textit{while} performing the computation (all while reusing the application’s code).
Haven’t analysed with respect to CPCA or ACPCA

- Implementation is not complete.
- Analysis is hard
  - Depends on program state.
  - Attacker’s goal needs to be defined.
  - Need tools to perform analysis.
We need rigorous evaluation of defenses.

We can have defenses that apply to all exploit techniques.
Questions?