Low-Level Software Security: Attacks and Defenses; Control Flow Integrity

System Security Seminar
Presenter: Cătălin Hrițcu
References

- *Control-flow integrity*, by Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti (CCS 2005)

- Many thanks to Úlfar Erlingsson from Microsoft Research and Reykjavk University
  - Many of the slides are from his FOSAD 2007 talk
  - Opinions and mistakes are still mine
A real-world attack

- Microsoft Windows animated cursor buffer overflow vulnerability (March 30, 2007)
Rated as “Extremely critical”
Exploited immediately in the wild

- Attack vectors:
  - HTML email / spam
  - Browsers vulnerable - 2000+ different sites hosting exploit
Microsoft reacts fast

- Releases “out of band” patch (April 3)

### Severity Ratings and Vulnerability Identifiers:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GDI Local Elevation of Privilege Vulnerability - <a href="#">CVE-2006-5758</a></td>
<td>Elevation of Privilege</td>
<td>Important</td>
<td>Important</td>
<td>Not Affected</td>
<td>Not Affected</td>
</tr>
<tr>
<td>WMF Denial of Service Vulnerability <a href="#">CVE-2007-1211</a></td>
<td>Denial of Service</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Not Affected</td>
</tr>
<tr>
<td>EMF Elevation of Privilege Vulnerability <a href="#">CVE-2007-1212</a></td>
<td>Elevation of Privilege</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>GDI Invalid Window Size Elevation of Privilege Vulnerability <a href="#">CVE-2006-5586</a></td>
<td>Elevation of Privilege</td>
<td>Important</td>
<td>Important</td>
<td>Not Affected</td>
<td>Not Affected</td>
</tr>
<tr>
<td>GDI Incorrect Parameter Local Elevation of Privilege Vulnerability - <a href="#">CVE-2007-1215</a></td>
<td>Elevation of Privilege</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Font Rasterizer Vulnerability - <a href="#">CVE-2007-1213</a></td>
<td>Elevation of Privilege</td>
<td>Important</td>
<td>Not Affected</td>
<td>Not Affected</td>
<td>Not Affected</td>
</tr>
</tbody>
</table>

### Aggregate Severity of All Vulnerabilities

- Critical
Low-level attacks

- Buffer overflows ~50% of all reported attacks
  - 1988: Robert Morris’s Internet Worm
  - 2000: Code Red, SQL Slammer
  - 2007: Windows .ANI vulnerability

- Possible whenever compiling a high-level language into a low-level one without guaranteeing that the translation preserves the high-level abstractions
The legacy of C

- Millions of lines of security-critical code written in C
- C compilers do not guarantee any property of the translation
- Even if we ignore compilation, C is not type and memory safe
  - This often leads to exploitable vulnerabilities
- Safer low-level languages: Cyclone, CCured, SafeC, ...
- Most of them are safe dialects of C
- Existing C code can be migrated with some changes
Promising safe C compiler from Berkeley

- Designed to work on existing real-world C code
  - Including the Linux kernel itself!
- Allows for incremental transition
  - Memory layout of data structures preserved
- Dependent types (additional annotations)
  - Hybrid type checking (static + dynamic)
- Low performance impact
  - Average of 10-20% for CPU-intensive tests
- Low annotation burden
  - For the Linux kernel less than 1% of lines annotated

Might be the subject of a future talk in this seminar
High-level languages

- e.g. Java, C#, Python, O’Caml, Standard ML, ...
- Usually promise to be safer and more secure
- Their actual safety still depends on low-level details
  - Compiler and runtime-system need to be correct
  - Static checking is not enough
    - Most high-level properties need to be enforced at run-time
      - Safety vs. performance tradeoff
- Rewriting legacy code in a new language is in most cases out of the question
  - Even if the target language would be perfectly secure
  - And the run-time overhead negligible
Mitigation techniques

- The main subject of this talk
- Limited language-based defenses that
  - Work on legacy code
  - Are fully automatic
  - Operate at the lowest level (machine-code)
  - Involve no source-code changes (at most re-compilation)
  - Typically, runtime checks to guarantee high-level properties
  - Unobtrusive: close to zero overhead and zero false positives
  - Only prevent certain vulnerabilities / attacks
    - Often unclear what vulnerabilities are covered
Mitigations are a compromise

- Mitigations are limited, correct software is better
  - Wouldn’t need any defenses if software was “correct”...

- So why not just fix all software?
  - Fixing software is difficult, costly, and error-prone
  - It is hard even to specify what “correct” should mean
  - Needs source, build environments, etc., and may interact badly with testing, debugging, deployment, and servicing

- Mitigations are not the optimal solution, but ...
  - They can rule out many practical attacks
  - Some of the ones we will see are deployed in practice (e.g. Windows XP SP2 and Vista)
Outline

- A simple buffer overflow example
- Two simple mitigation techniques
  - Stack canaries and cookies
  - Preventing data execution (NXD)
- A more complex jump-to-libc attack
- A more powerful mitigation technique
  - Control-flow integrity
A concrete stack overflow example

```c
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

- Attack overflows a (fixed-size) array on the stack
- The function return address points to the attacker’s code
- The best known low-level attack
  - Used by the Internet Worm in 1988 and commonplace since
A concrete stack overflow example

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037; argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f; argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00401263; return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x0012ff7c; saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0x00000072; tmp continues 'r'</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x61626f6f; tmp continues 'o'</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x662f2f3a; tmp continues ':'</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966; tmp array: 'f'</td>
</tr>
</tbody>
</table>

- The above stack snapshot is normal w/o overflow
- The arguments here are “file://” and “foobar”
A concrete stack overflow example

- A stack snapshot with a benign overflow

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
<td>argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
<td>argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00666473</td>
<td>return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x61666473</td>
<td>saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0x61666473</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x61666473</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x612f2f3a</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966</td>
<td>tmp array:</td>
</tr>
</tbody>
</table>

- In the above, the stack has been corrupted
- The second (attacker-chosen) arg is “asdfsdfsdfsdfsdfsdf”
- Of course, an attacker might not corrupt in this way...
A concrete stack overflow example

Now, a stack snapshot with a malicious overflow:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037 ; argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f ; argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x0012ff48 ; return address: address of attack payload</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0xXXXXXXXXXX ; irrelevant</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0xXXXXXXXXXX ; irrelevant</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0xfeeb2ecd ; attack payload</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0xXX2f2f3a ; tmp continues</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966 ; tmp array: 'f' 'i' 'l' 'e'</td>
</tr>
</tbody>
</table>

- In the above, the stack has been corrupted maliciously
- The args are “file://” and particular attacker-chosen data
- XX can be any non-zero byte value
Stack canaries

- Very simple defense
  - Assume a contiguous buffer overflow is used by attackers
  - And that the overflow is based on zero-terminated strings
  - Put canary with “terminator” values below the return address

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00401263</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x0012ff7c</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x00000072</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x61626f6f</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x662f2f3a</td>
</tr>
<tr>
<td>0x0012ff3c</td>
<td>0x656c6966</td>
</tr>
</tbody>
</table>

- Check canary integrity before using return address!
Stack cookies

- Can also use random, secret values: **cookies**
- Defends against non-null-terminated overflows (e.g. via memcpy)

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
<td>argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
<td>argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00401263</td>
<td>return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x0012ff7c</td>
<td>saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td><strong>0xF00DFEED</strong></td>
<td>a secret, random cookie value</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x00000072</td>
<td>tmp continues <code>r' \0 \0 \0</code></td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x61626f6f</td>
<td>tmp continues <code>o' </code>o' <code>b' </code>a'</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x662f2f3a</td>
<td>tmp continues <code>:</code> <code>/</code> <code>/</code> <code>f</code></td>
</tr>
<tr>
<td>0x0012ff3c</td>
<td>0x656c6966</td>
<td>tmp array: <code>f</code> <code>i</code> <code>l</code> <code>e</code></td>
</tr>
</tbody>
</table>

- Check cookie integrity before using return address!
- Implemented in Windows (/GS compiler flag)
Stack canaries and cookies

- Stack canaries and stack cookies have very little cost
  - Only needed on functions with local arrays
  - Even so, not always applied: heuristics determine when
    - (Not a good idea, as shown by recent ANI attack on Vista)

- Widely implemented: /GS, StackGuard, ProPolice, etc.
  - Implementations typically combine with other defenses

- Main limitations:
  - Only protects against contiguous stack-based overflows
  - No protection if attack happens before function returns
  - For example, must protect function-pointer arguments
  - Do not prevent heap-based buffer overflows
Preventing data execution

- Simply prevent the execution of data as code
- This prevents both stack and heap-based attacks
- There is hardware support for this (NX bit on x86)
  - Can be done with pretty much zero overhead
  - But it breaks a lot of software:
    - Most Win32 GUI apps, CLR (and JITs)
- Can also be done in software (SMAC)
- Limitations:
  - Attackers don’t always have to execute data as code
    - They can just corrupt data: data-only attacks
    - They can simply execute existing code: jump-to-libc
Jump-to-libc

- Any existing code can be executed by attackers
  - May be an existing function, such as `system()`
  - E.g., a function that is never invoked (dead code)
  - Or code in the middle of a function

- Can even be “opportunistic” code
  - Found within executable pages (e.g. switch tables)
  - Or found within existing instructions (long x86 instructions)

- Typically a step towards running attackers own shellcode

- These are *jump-to-libc* or *return-to-libc* attacks
- Allow attackers to overcome NX defenses
A new function to be attacked

- Computes the median integer in an input array
- Sorts a copy of the array and return the middle integer

```c
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) ); // copy the input integers
    qsort( tmp, len, sizeof(int), cmp ); // sort the local copy
    return tmp[len/2];                    // median is in the middle
}
```

- If len is larger than MAX_INTS we have a stack overflow
An example bad function pointer

- Many ways to attack the median function
- The cmp pointer is used before the function returns
  - It can be overwritten by a stack-based overflow
  - And stack canaries or cookies are not a defense
- Using jump-to-libc, an attack can also foil NX
- Use existing code to install and jump to attack payload
  - Including marking the shellcode bytes as executable
- Example of indirect code injection
- (As opposed to direct code injection in previous attacks)
Concrete jump-to-libc attack example

- A normal stack for the median function
- Stack snapshot at the point of the call to memcpy
- MAX_INTS is 8
- The tmp array is empty, or all zero
Concrete jump-to-libc attack example

- A **benign** stack overflow in the median function

- Not the values that an attacker will choose ...

<table>
<thead>
<tr>
<th>stack address</th>
<th>benign overflow contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff38</td>
<td>0x1111110d; cmp argument</td>
</tr>
<tr>
<td>0x0012ff34</td>
<td>0x1111110c; len argument</td>
</tr>
<tr>
<td>0x0012ff30</td>
<td>0x1111110b; data argument</td>
</tr>
<tr>
<td>0x0012ff2c</td>
<td>0x1111110a; return address</td>
</tr>
<tr>
<td>0x0012ff28</td>
<td>0x11111109; saved base pointer</td>
</tr>
<tr>
<td>0x0012ff24</td>
<td>0x11111108; tmp final 4 bytes</td>
</tr>
<tr>
<td>0x0012ff20</td>
<td>0x11111107; tmp continues</td>
</tr>
<tr>
<td>0x0012ff1c</td>
<td>0x11111106; tmp continues</td>
</tr>
<tr>
<td>0x0012ff18</td>
<td>0x11111105; tmp continues</td>
</tr>
<tr>
<td>0x0012ff14</td>
<td>0x11111104; tmp continues</td>
</tr>
<tr>
<td>0x0012ff10</td>
<td>0x11111103; tmp continues</td>
</tr>
<tr>
<td>0x0012ff0c</td>
<td>0x11111102; tmp continues</td>
</tr>
<tr>
<td>0x0012ff08</td>
<td>0x11111101; tmp buffer starts</td>
</tr>
<tr>
<td>0x0012ff04</td>
<td>0x00000040; memcpy length argument</td>
</tr>
<tr>
<td>0x0012ff00</td>
<td>0x00353050; memcpy source argument</td>
</tr>
<tr>
<td>0x0012ffec</td>
<td>0x0012ff08; memcpy destination arg.</td>
</tr>
</tbody>
</table>
Concrete jump-to-libc attack example

- A malicious stack overflow in the median function
- The attack doesn’t use the return address (e.g., to avoid stack canary or cookie defenses)
- Control-flow is redirected in qsort
- Uses jump-to-libc to foil NX defenses
Concrete jump-to-libc attack example

- Below shows the context of cmp invocation in qsort
- Goes to a 4-byte *trampoline* sequence found in a library

```
...  
push   edi ; push second argument to be compared onto the stack
push   ebx ; push the first argument onto the stack
call   [esp+comp_fp] ; call comparison function, indirectly through a pointer
add    esp, 8 ; remove the two arguments from the stack
test   eax, eax ; check the comparison result
jle    label_lessthan ; branch on that result
...
```

<table>
<thead>
<tr>
<th>address</th>
<th>opcode_bytes</th>
<th>assembly-language version of the machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7c971649</td>
<td>0x8b 0xe3</td>
<td>mov esp, ebx ; change the stack location to ebx</td>
</tr>
<tr>
<td>0x7c97164b</td>
<td>0x5b</td>
<td>pop ebx ; pop ebx from the new stack</td>
</tr>
<tr>
<td>0x7c97164c</td>
<td>0xc3</td>
<td>ret ; return based on the new stack</td>
</tr>
</tbody>
</table>
The intent of the jump-to-libc attack

- Perform a series of calls to existing library functions
- With carefully selected arguments

```c
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51

// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd); // function at 0x7c80978e

// invoke the four bytes of attack payload machine code
((void (*)())0x70000000)(); // payload at 0x70000000

- The effect is to install and execute the attack payload
x86 __cdecl function-call convention

- Push parameters onto the stack, from right to left
- Call the function
- Save and update the %ebp
- Allocate local variables
- Perform the function's purpose
- Release local storage
- Restore saved registers
- Restore the old base pointer
- Return from the function
  - The RET instruction pops the old %EIP from the stack and jumps to that location. This gives control back to the caller function. Only the stack pointer and instruction pointers are modified by a subroutine return.
- Clean up pushed parameters
How the attack unwinds the stack

- First invalid control-flow edge goes to trampoline
- Trampoline returns to the start of VirtualAlloc
- Which returns to the start of the InterlockedExch. function
- Which returns to the copy of the attack payload

<table>
<thead>
<tr>
<th>stack address</th>
<th>malicious overflow contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff38</td>
<td>0x7c971649</td>
</tr>
<tr>
<td>0x0012ff34</td>
<td>0x1111110c</td>
</tr>
<tr>
<td>0x0012ff30</td>
<td>0x1111110b</td>
</tr>
<tr>
<td>0x0012ff2c</td>
<td>0xfeeb2ecd</td>
</tr>
<tr>
<td>0x0012ff28</td>
<td>0x70000000</td>
</tr>
</tbody>
</table>

New executable copy of attack payload
Interlocked Exchange
VirtualAlloc
Where to find useful trampolines?

- In Linux libc, one in 178 bytes is a 0xc3 ret opcode
- One in 475 bytes is an opportunistic, or unintended, ret

```
f7 c7 07 00 00 00    test   edi, 0x00000007
0f 95 45 c3          setnz  byte ptr [ebp-61]
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f    movl   edi, 0x0f000000
95                     xchg   eax, ebp
45                      inc    ebp
 c3                     ret
```

- All of these may be useful somehow
Generalized jump-to-libc attacks

- Recent demonstration by Shacham [upcoming CCS’07]
  - Possible to achieve anything by only executing trampolines
  - Can compose trampolines into “gadget” primitives
  - Such “return-oriented-computing” is Turing complete
  - Practical, even if only opportunistic ret sequences are used

- Confirms a long-standing assumption:
  
  *if arbitrary jumping around within existing, executable code is permitted*

  *then*

  *an attacker can cause any desired, bad behavior*
Jump-to-libc attacks

- Jump-to-libc attacks are of great practical concern
  - For instance, recent ANI attack on Vista is similar to median

- Traditionally, return-to-libc with the target system()
  - Removing system() is neither a good nor sufficient defense
    - Generality of trampolines makes this a unarguable point
  - Anyway difficult to eliminate code from shared libraries

- Based on knowledge of existing code, and its addresses
  - Attackers must deal with natural software variability
  - Increasing the variability can be a good defense

- Best defense is to lock down the possible control flow
  - Other, simpler measures will also help
~60% of attacks subvert the expected control flow
- Enforcing control-flow integrity would prevent such attacks
Assumptions about control flow

- We write our code in high-level languages

- Naturally, our execution model assumes:
  - Functions start at the beginning
  - They (typically) execute from beginning to end
  - And, when done, they return to their call site
  - Only the code in the program can be executed
  - The set of executable instructions is limited to those output during compilation of the program
Assumptions about control flow

- We write our code in high-level languages

- But, actually, at the level of machine code
  - Can start in the middle of functions
  - A fragment of a function may be executed
  - Returns can go to any program instruction
  - All the data has usually been executable
  - On the x86, can start executing not only in the middle of functions, but middle of instructions!
What bytes will the CPU interpret?

- Hardware places few constrains on control flow
- A call to a function-pointer can lead many places:

![Possible Execution of Memory](image)

- Possible control flow destination
- Safe code/data

Data memory

Code memory for function A

Code memory for function B

x86

x86/NX

x86/CFI
Enforcing control-flow integrity

- Only certain control-flow is possible in software
  - Even in C there are function and expression boundaries
  - Should also consider who-can-go-where, and dead code

- Control-flow integrity means that execution proceeds according to a specified control-flow graph (CFG).
  - Reduces gap between machine code and high-level languages

- Can enforce with CFI mechanism, which is simple, efficient, and applicable to existing software.
  - CFI enforces a basic property that thwarts a large class of attacks— without giving “end-to-end” security.
Guards for control flow integrity

- CFI guards restrict computed jumps and calls
  - Calls through function pointers (e.g. virtual methods in C++)
  - All return, exception and switch statements
- Direct calls are unaffected
- CFI guard matches label at source and target
  - Labels are constants embedded in machine-code
  - Labels are not secret, but must be unique
- Two destinations are equivalent when the CFG contains edges to it from the same set of sources
  - Equivalent destinations are labeled the same
  - i.e. a label uniquely identifies a CFG equivalence class
A simple example

```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 );
}
```

- Ensure “labels” are correct at load- and run-time
  - Bit patterns identify different points in the code
  - Indirect control flow must go to the right pattern
- Can be enforced using software instrumentation
  - Even for existing, legacy software
Overview of a system with CFI

- **Machine code rewriting using instrumentation tool**
  - Applies to legacy Windows x86 executables
  - Code rewriting need not be trusted, because of the verifier
  - The verifier is simple (2 KLoC, mostly parsing x86 opcodes)
CFI formal study [ICFEM’05]

- Formally validated the benefits of CFI
  - Defined a machine code semantics
  - Powerful attacker model
    - Attacker can arbitrarily control all of data memory
  - Proved that, with CFI, execution always follows the CFG, even when under attack

- Assumptions
  - NXD: Data cannot be executed (hardware or software)
  - NWC: Code cannot be modified (hardware, already used)
  - We can rely on values in distinguished registers
  - Jumps cannot go into the middle of instructions
    - A convenient simplification to make the proof manageable
CFI as foundation for other prop.

- CFI can be used as a foundation for efficiently enforcing more sophisticated security properties

  - Software fault isolation (e.g. sandboxing)
    - Dynamically check memory accesses to emulate traditional memory protection

- Software memory access control
  - Stronger than software fault isolation: isolated data memory regions accessible only from particular code
  - Removes NXD assumption, but adds extra overhead

- Protected shadow call stack
  - ID checks on return replaced by the use of a call stack
  - Very little extra overhead (at least with x86-specific tricks)
Cost and Benefits

- CFI overhead: ~16% in synthetic CPU-bound benchmarks
  - Is this really unobtrusive (close to zero overhead)?
- Effectively stops most jump-to-libc attacks
  - No trampolining, even if CFI enforces a very coarse CFG
  - E.g., may have two labels -- for call sites and start of functions
- Limitation: Data-only attacks
Conclusion

- Mitigation techniques
  - Automatic defenses that work on legacy code
  - Operate at the machine-code level
  - Involve no source-code changes
  - Have close to zero overhead
  - Only prevent certain kinds of attacks
  - May provide a false feeling of security ... like a Volvo
  - Are not substitutes for correct code or safer languages
  - Do not protect against denial-of-service attacks

- Control-flow integrity
  - Particularly powerful mitigation technique
  - Prevents many kinds of attacks, including jump-to-libc