Low-level software security
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.
Flaws

• Circumvention and hijacking are common in security in many realms.
  – Tanks drive around fortifications.
  – Robbers bribe bank guards.

• In computer systems, they are sometimes the consequence of design weaknesses.

• But many result from implementation flaws: small but catastrophic errors in code.
Software security

Software security is
• not only about implementation flaws,
• not only about low-level attacks and defenses,
• certainly not only about buffer overflows,
but low-level attacks and defenses
• remain important,
• illustrate themes and techniques that appear throughout software systems.
An example
An example

```c
int f(int x, char y) {
    char t[16] ;
    initialize(t) ;
    t[x] = y ;
    return 0 ;
}
```
An example

```c
int f(int x, char y) {
    char t[16];
    initialize(t);
    t[x] = y;
    return 0;
}
```
So what?

• Threat model: The attacker chooses inputs.
⇒ The attacker can (try to) modify a location of their choice at some offset from t’s address.

• Some possible questions:
  – Can the attacker find the vulnerability and call f?
  – Can the attacker identify good target locations?
  – Can the attacker predict t’s address?
  – Will the exploit work reliably? cause crashes?
Going further: two examples
[from Chen, Xu, Sezer, Gauriar, and Iyer]

• Attack NULL-HTTPD (a Web server on Linux).
  – POST commands can trigger a buffer overflow.
Change the configuration string of the CGI-BIN path:
  – The mechanism of CGI:
    • Server name = www.foo.com
    • CGI-BIN = /usr/local/httpd/exe
    • Request URL = http://www.foo.com/cgi-bin/bar
      → Normally, the server runs /usr/local/httpd/exe/bar
  – An attack:
    • Exploiting the buffer overflow, set CGI-BIN = /bin
    • Request URL = http://www.foo.com/cgi-bin/sh
      → The server runs /bin/sh

⇒ The attacker gets a shell on the server.
• Attack SSH Communications SSH Server:

```c
void do_authentication(char *user, ...) {
    int auth = 0; /* initially auth is false */
    ...
    while (!auth) {
        /* Get a packet from the client */
        type = packet_read(); /* has overflow bug */
        switch (type) { /* can make auth true */
            ...
            case SSH_CMSG_AUTH_PASSWORD:
                if (auth_password(user, password))
                    auth = 1;
            case ...
        }
        if (auth) break;
    }
    /* Perform session preparation. */
    do_authenticated(...);
}
⇒ The attacker circumvents authentication.
```
The attacker circumvents authentication.

These are **data-only** attacks.

- The most classic attacks often inject code.
- Injecting code is also central in higher-level attacks such as SQL injection and XSS.
Run-time protection: the arms race

- Many attack methods:
  - Buffer overflows
  - Jump-to-libc exploits
  - Use-after-free exploits
  - Exception overwrites
  - ...

- Many defenses:
  - Stack canaries
  - Safe exception handling
  - NX data
  - Layout randomization
  - ...

- Not necessarily perfect in a precise sense
- Nor all well understood
- But useful mitigations
New Windows zero-day surfaces as researcher releases attack code

SMB bug could be exploited on Windows XP, Server 2003 to hijack machines, say experts

By Gregg Keizer
February 15, 2011 03:59 PM ET

Secunia added that a buffer overflow could be triggered by sending a too-long Server Name string in a malformed Browser Election Request packet. In this context, "browser" does not mean a Web browser, but describes other Windows components which access the OS' browser service.
A buffer overflow

define function \( f(\text{arg}) = \)

let \( t \) be a local variable of size \( n \);

\( \) copy contents of \( \text{arg} \) into \( t \);

\( \) ...

• The expectation is that the contents of \( \text{arg} \) is at most of size \( n \).
A buffer overflow

define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
    ...

• The expectation is that the contents of arg is at most of size n.
• In memory, we would have:

    local variable t  return address

    First  ...  (nothing yet)  f’s caller address  ...
A buffer overflow

define function f(arg) =
   let t be a local variable of size n;
   copy contents of arg into t;
   ...

• The expectation is that the contents of arg is at most of size n.
• In memory, we would have:

<table>
<thead>
<tr>
<th>local variable t</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>...</td>
</tr>
<tr>
<td>Later</td>
<td>...</td>
</tr>
</tbody>
</table>
A buffer overflow

define function $f(\text{arg}) =$
  let $t$ be a local variable of size $n$;
  copy contents of $\text{arg}$ into $t$;
  ...

• If this size is too big and not checked (either statically or dynamically), there can be trouble.
A buffer overflow

define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
    ...

• If this size is too big and not checked (either statically or dynamically), there can be trouble.
• In memory, we could have:

<table>
<thead>
<tr>
<th>local variable t</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(nothing yet)</td>
<td>f’s caller address</td>
</tr>
</tbody>
</table>

First
A buffer overflow

define function \( f(\text{arg}) = \)
let \( t \) be a local variable of size \( n \);
copy contents of \( \text{arg} \) into \( t \);

\[ \ldots \]

- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could have:

<table>
<thead>
<tr>
<th>First</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(nothing yet)</td>
<td>( f )'s caller address</td>
</tr>
<tr>
<td>Later</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{arg contents} )</td>
<td>(part)</td>
</tr>
</tbody>
</table>
A buffer overflow

define function \( f(\text{arg}) = \)
let \( t \) be a local variable of size \( n \);
copy contents of \( \text{arg} \) into \( t \);

...  

- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:

<table>
<thead>
<tr>
<th>First</th>
<th>...</th>
<th>(nothing yet)</th>
<th>f’s caller address</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Later</td>
<td>...</td>
<td>arg contents</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow

define function f(arg) =
   let t be a local variable of size n;
   copy contents of arg into t;
   ...

- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:

<table>
<thead>
<tr>
<th>local variable t</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First</th>
<th></th>
<th>(nothing yet)</th>
<th>f’s caller address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Later</td>
<td></td>
<td>arg contents = ...</td>
<td>new return address ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A buffer overflow

```plaintext
define function \( f(arg) = \)
  let \( t \) be a local variable of size \( n \);
  copy contents of \( arg \) into \( t \);
  ...
```

- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:

<table>
<thead>
<tr>
<th>local variable ( t )</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>...</td>
</tr>
<tr>
<td>Later</td>
<td>...</td>
</tr>
</tbody>
</table>
A buffer overflow

define function f(arg) =
let t be a local variable of size n;
copy contents of arg into t;
...

• If this size is too big and not checked (either statically or dynamically), there can be trouble.

• In memory, we could also have:

<table>
<thead>
<tr>
<th>local variable t</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First

<table>
<thead>
<tr>
<th>...</th>
<th>(nothing yet)</th>
<th>f’s caller address</th>
<th>...</th>
</tr>
</thead>
</table>

Later

<table>
<thead>
<tr>
<th>...</th>
<th>arg contents = ...</th>
<th>new return address + code</th>
<th>...</th>
</tr>
</thead>
</table>
Stack canaries and cookies

**define function** \( f(\text{arg}) = \)

let \( t \) be a local variable of size \( n \);
copy contents of \( \text{arg} \) into \( t \);  
...

- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.

<table>
<thead>
<tr>
<th></th>
<th>local variable ( t )</th>
<th>canary</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>...</td>
<td>(nothing yet)</td>
<td>“tweety”</td>
</tr>
</tbody>
</table>
Stack canaries and cookies

define function $f(\text{arg}) =$
let $t$ be a local variable of size $n$;
copy contents of $\text{arg}$ into $t$;
...

- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.

<table>
<thead>
<tr>
<th>local variable $t$</th>
<th>canary</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>(nothing yet)</td>
<td>“tweety”</td>
</tr>
<tr>
<td>$\text{arg contents} = \ldots$</td>
<td>new return address + code</td>
<td>$f$'s caller address</td>
</tr>
</tbody>
</table>
There are more things

- Stack canaries and cookies can be effective in impeding many buffer overflows on the stack.

But:
- They need to be applied consistently.
- Sometimes they are judged a little costly.
- They do not help if corrupted data (e.g., a function pointer) is used before the return.
- And there are many kinds of overflows, and many other kinds of vulnerabilities.
NX (aka DEP)

Many attacks rely on injecting code.

$\Rightarrow$ *So a defense is to require that data that is writable cannot be executed.*

- This requirement is supported by mainstream hardware (e.g., x86 processors).
NX (aka DEP)

Many attacks rely on injecting code.

⇒ So a defense is to require that data that is writable cannot be executed.*

• This requirement is supported by mainstream hardware (e.g., x86 processors).

* An exception must be made in order to allow compilation (e.g., JIT compilation for JavaScript).
What bytes will the CPU interpret?

- Mainstream hardware typically places few constraints on control flow.
- A call can lead to many places:

  - Possible control-flow destination
  - Safe code/data

Data memory

Code memory for function A

Code memory for function B

x86

x86/NX

RISC/NX
Executing existing code

• With NX defenses, attackers cannot simply inject data and then run it as code.
• But attackers can still run existing code:
  – the intended code in an unintended state,
  – an existing function, such as `system()`,
  – even dead code,
  – even code in the middle of a function,
  – even “accidental” code (e.g., starting half-way in a long x86 instruction).
An example of accidental x86 code
[Roemer et al.]

Two instructions in the entry point ecb_crypt are encoded as follows:

\[\begin{array}{ll}
\text{f7 c7 07 00 00 00} & \text{test } 0x00000007, \%edi} \\
\text{0f 95 45 c3} & \text{setnzb } -61(\%ebp) \\
\end{array}\]

Starting one byte later, the attacker instead obtains

\[\begin{array}{ll}
\text{c7 07 00 00 00 0f} & \text{movl } 0x0f00000000, (\%edi) \\
\text{95} & \text{xchg } \%ebp, \%eax \\
\text{45} & \text{inc } \%ebp \\
\text{c3} & \text{ret}
\end{array}\]
Layout randomization

Attacks often depend on addresses.

Let us randomize the addresses!

– Considered for data at least since the rise of large virtual address spaces (e.g., [Druschel & Peterson, 1992] on fbufs).

– Now present in Linux (PaX), Windows, Mac OS X, iOS, Android (4.0).
Implementations

• The randomization can be performed at build, install, boot, or load time.
Implementations

• The randomization can be performed at build, install, boot, or load time.
Implementations

• The randomization can be performed at build, install, boot, or load time.
• It may be at various granularities.
• It need not have performance cost, but it may complicate compatibility.
A theory of layout randomization
[with Gordon Plotkin, now Jérémy Planul]

• Define *high-level programs*, with symbolic locations (e.g., $l := 3$), and *low-level programs*, with numbers as addresses (e.g., $8686 := 3$).
  ➔ View randomization as part of a translation.
A theory of layout randomization
[with Gordon Plotkin, now Jérémie Planul]

• Define *high-level programs*, with symbolic locations (e.g., $l := 3$), and *low-level programs*, with numbers as addresses (e.g., $8686 := 3$).

  → View randomization as part of a translation.

• View attackers as contexts, i.e., other programs with which our programs interact.

  → Relate low-level contexts to high-level contexts.
A theory of layout randomization
[with Gordon Plotkin, now Jérémy Planul]

• Define *high-level programs*, with symbolic locations (e.g., \( l := 3 \)), and *low-level programs*, with numbers as addresses (e.g., \( 8686 := 3 \)).
  → View randomization as part of a translation.

• View attackers as contexts, i.e., other programs with which our programs interact.
  → Relate low-level contexts to high-level contexts.

• Phrase security properties as equivalences.
  → Study whether equivalences are preserved.
The source language

• Higher-order lambda calculus,
• with read/write/execute operations on locations that hold natural numbers,
• with standard base types and optionally a type of locations,
• also sometimes with an error constant (which we assume here).
Syntax

- Types:

\[
\sigma ::= b \mid \text{unit} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \rightarrow \sigma
\]

where \(b\) ranges over basic types which always include \texttt{nat} and may include \texttt{loc}.
Syntax (cont.)

- Programs:

\[ M ::= x \mid c \mid \ast \mid (M, M) \mid \text{fst } M \mid \text{snd } M \mid \text{inl}_{\sigma,\sigma} M \mid \text{inr}_{\sigma,\sigma} M \mid \text{cases } M \text{ inl } x:\sigma. M \text{ inr } x:\sigma. M \mid \lambda x:\sigma. M \mid MM \mid \text{rec}(f:\sigma \rightarrow \tau, x:\sigma). M \]

where \( c \) ranges over constants, each of a unique type. These include the natural numbers, the usual arithmetic operations, constants for memory access (e.g., \( \text{run, :=} \)), and constants for raising errors.
Memory access
(some specifics)

• Memory-access constants:

\[ l : \text{loc} \ (l \in \text{Loc}) \]
\[ !_{\text{loc}} : \text{loc} \rightarrow \text{nat} \]
\[ :=_{\text{loc}} : \text{loc} \times \text{nat} \rightarrow \text{unit} \]
\[ \text{run}_{\text{loc}} : \text{loc} \rightarrow \text{unit} \]

• Some semantics:

\[(s, !_{\text{loc}} l) \rightarrow (s, n) \quad (\text{if } s(l) = n)\]
\[(s, l :=_{\text{loc}} n) \rightarrow (s[l \mapsto n], \star) \quad (\text{if } l \in \text{DataLoc})\]
\[(s, \text{run}_{\text{loc}} l) \rightarrow (s', \star) \quad (\text{if } l \in \text{CodeLoc}, s(l) = n, s' = Dc(n)(s))\]

where a store \( s \) is a function from Loc to natural numbers, and \( Dc \) is an “instruction decoding” function.
The target language

• Much like the source language,
• but with natural-number addresses rather than locations.

\[ l : \text{nat} \quad (\text{for } l \in \text{Loc}) \]
\[ !_{\text{nat}} : \text{nat} \to \text{nat} \]
\[ :=_{\text{nat}} : \text{nat} \times \text{nat} \to \text{unit} \]
\[ \text{run}_{\text{nat}} : \text{nat} \to \text{unit} \]
The target model(s), informally

- A **layout** $w$ is a function $\text{Loc} \rightarrow \{0, \ldots, c\}$ chosen at random (for instance, uniformly).
- A **memory** $m$ is a function: $\{0, \ldots, c\} \rightarrow \mathbb{N} + 1$
  - Memory may be accessed directly through natural-number addresses.
  - Some addresses may be unused.
- Accesses to unused addresses are either **fatal errors** or **recoverable errors**.
  - These two variants both make sense, but lead to different results.
Attackers as contexts

• A **public program** is one that cannot access private locations directly. I.e.:
  – Our languages have constants for locations (**Loc**).
  – We distinguish sets of **public** locations (**PubLoc**) and **private** locations (**PriLoc**).
  – Private ones cannot occur in public programs.

• For us, attackers are public contexts.
Equivalences

In the source language, two programs are **publically equivalent** if no public context can distinguish them:

for $M, N$ of the same type $\sigma$, \[ M \approx_{h,p} N \]
iff for every initial store $s$, every public $C$ of type $\sigma \rightarrow \text{bool}$
(1) $CM$ and $CN$ both diverge,
(2) or they both give an error,
(3) or they both yield the same result value and two new stores that coincide on PubLoc.

In the target language, $M \approx_{l,p} N$ is similar, but with probabilities (over the choice of layout).
Equivalences (cont.)

Secrecy and integrity properties can be phrased as public equivalences.

E.g., for a private location \( l \)

\[
\begin{array}{l}
l := c \quad \approx_{h,p} \quad l := c' \\
\end{array}
\]

\[
\begin{array}{l}
\lambda f : \text{nat} \rightarrow \text{unit}.
\quad l := c; \\
\quad f(c); \\
\quad \text{if } l = c \text{ then } l' := c \text{ else } l' := c' \\
\end{array}
\]

\[
\begin{array}{l}
\lambda f : \text{nat} \rightarrow \text{unit}.
\quad l := c; \\
\quad f(c); \\
\quad l' := c \\
\end{array}
\]

\[
\begin{array}{l}
\approx_{h,p} \\
\end{array}
\]
Preserving equivalences
(“full abstraction”)

With each high-level program $M$ we associate a low-level program $\downarrow M$.

**Theorem:** Suppose that $M$ and $N$ are high-level terms of type $\sigma$. Assume that $\sigma$ is loc-free.

If $M \approx_{h,p} N$ then $\downarrow M \approx_{l,p} \downarrow N$.
Layout randomization depends on secrecy, but...

• The secrecy is not always strong.
  – E.g., there cannot be much address randomness on 32-bit machines.
  – E.g., low-order address bits may be predictable.

• The secrecy is not always well-protected.
  – Pointers may be disclosed.
  – Functions may be recognized by their behavior.
Layout randomization depends on secrecy, but...

• This secrecy is not always effective.
  – “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.

Browser

A nice Web site that attracts traffic (owned by the attacker)
Layout randomization depends on secrecy, but...

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.

Browser

A nice Web site that attracts traffic (owned by the attacker)
Layout randomization depends on secrecy, but...

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
Layout randomization depends on secrecy, but…

• This secrecy is not always effective.
  – “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
Layout randomization depends on secrecy, but...

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
Layout randomization depends on secrecy, but...

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.

![Diagram of heap-spray area and browser with JavaScript exploit code](image)
Layout randomization depends on secrecy, but...

• This secrecy is not always effective.
  – “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.

A nice Web site that attracts traffic
(owned by the attacker)

Web page with JavaScript

Jump (e.g., via buffer overflow)

Browser

Heap-spray area

NOP slide Exploit code

NOP slide Exploit code

NOP slide Exploit code

NOP slide Exploit code

fill
Layout randomization depends on secrecy, but...

• This secrecy is not always effective.
  – “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.

<table>
<thead>
<tr>
<th>Date</th>
<th>Browser</th>
<th>Description</th>
<th>milw0rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/2004</td>
<td>IE</td>
<td>IFRAME Tag BO</td>
<td>612</td>
</tr>
<tr>
<td>04/2005</td>
<td>IE</td>
<td>DHTML Objects Corruption</td>
<td>930</td>
</tr>
<tr>
<td>01/2005</td>
<td>IE</td>
<td>.ANI Remote Stack BO</td>
<td>753</td>
</tr>
<tr>
<td>07/2005</td>
<td>IE</td>
<td>javaprxy.d11 COM Object</td>
<td>1079</td>
</tr>
<tr>
<td>03/2006</td>
<td>IE</td>
<td>createTextRang RE</td>
<td>1606</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>VML Remote BO</td>
<td>2408</td>
</tr>
<tr>
<td>03/2007</td>
<td>IE</td>
<td>ADODB Double Free</td>
<td>3577</td>
</tr>
<tr>
<td>09/2006</td>
<td>IE</td>
<td>WebViewFolderIcon setSlice</td>
<td>2448</td>
</tr>
<tr>
<td>09/2005</td>
<td>FF</td>
<td>0xAD Remote Heap BO</td>
<td>1224</td>
</tr>
<tr>
<td>12/2005</td>
<td>FF</td>
<td>compareTo() RE</td>
<td>1369</td>
</tr>
<tr>
<td>07/2006</td>
<td>FF</td>
<td>Navigator Object RE</td>
<td>2082</td>
</tr>
<tr>
<td>07/2008</td>
<td>Safari</td>
<td>Quicktime Content-Type BO</td>
<td>6013</td>
</tr>
</tbody>
</table>

Source: Ratanaworabhan, Livshits, and Zorn (2009)
Layout randomization depends on secrecy, but...

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
  - “Heap feng shui” influences heap layout [Sotirov].
  - ...

Layout randomization: status

This is an active area, with

• variants and ongoing improvements to the randomization and its application,

• variants of the attacks,

• techniques detecting or mitigating the attacks.

Overall, randomization is widespread and seems quite effective but not a panacea.
Diverting control flow

• Many attacks cause some sort of subversion of the expected control flow.

  – E.g., an argument that is “too large” can cause a function to jump to an unexpected place.

• Several techniques prevent or mitigate the effects of many control-flow subversions.
  – E.g., canaries help prevent some bad returns.
Control-flow integrity (CFI)

- CFI means that execution proceeds according to a specified control-flow graph (CFG).
- CFI is a basic property that thwarts a large class of attacks.
What bytes will the CPU interpret, with CFI?

• E.g., we may allow jumps to the start of any function (defined in a higher-level language):
What bytes will the CPU interpret, with CFI? (cont.)

• Or we may allow jumps the start of B only from a particular call site in A:

```plaintext
Possible control-flow destination
Safe code/data
Data memory
Code memory for function A
Code memory for function B
```

- x86
- x86/NX
- RISC/NX
- x86/CFI
Some implementation strategies for CFI

1. A fast interpreter performs control-flow checks (“Program Shepherding”).

2. A compiler emits code with control-flow checks (as in WIT).

3. A code rewriter adds control-flow checks (as in PittSFIeld, where all control-flow targets are required to end with two 0s).
A rewriting-based system
[with Budiu, Erlingsson, Ligatti, Peinado, Necula, and Vrable]

- The rewriting inserts guards to be executed at run-time, before control transfers.
- It need not be trusted, because of the verifier.
Example

- Code uses data and function pointers,
- susceptible to effects of memory corruption.

```c
int foo(fptr pf, int* pm) {
    int err;
    int A[4];
    // ...
    pf(A, pm[0], pm[1]);
    // ...
    if( err ) return err;
    return A[0];
}
```

Machine-code basic blocks:

- ECX := Mem[ESP + 4]
- EDX := Mem[ESP + 8]
- ESP := ESP - 0x14
- push Mem[EDX + 4]
- push Mem[EDX]
- push ESP + 8
- call ECX
- push Mem[EDX + 4]
- push Mem[EDX]
- push ESP + 8
- call ESP + 8
- EAX := Mem[ESP + 0x10]
- if EAX != 0 goto L
- EAX := Mem[ESP]
- L: ... and return
Example (cont.)

- We add guards for checking control transfers.
- These guards are “inline reference monitors”.

```c
int foo(fptr pf, int* pm) {
    int err;
    int A[4];
    // ...
    pf(A, pm[0], pm[1]);
    // ...
    if( err ) return err;
    return A[0];
}
```
A CFI guard
(a simple variant)

- A CFI guard matches IDs at source and target.
  - IDs are constants embedded in machine code.
  - IDs are not secret, but must be unique.

pf(A, pm[0], pm[1]);
// ...  

C source code

Machine code with 0x12345678 as CFI guard ID

...  
EAX := 0x12345678  
if Mem[ECX-4] != EAX goto ERR  
call ECX  

// ...  
ret
Proving that CFI works

• Some of the recent systems come with (and were guided by) proofs of correctness.

• The basic steps may be:
  1. Define a machine language and its semantics.
  2. Define when a program has appropriate instrumentation, for a given control-flow graph.
  3. Prove that all executions of programs with appropriate instrumentation follow the prescribed control-flow graphs.
1. A small model of a machine

- Instructions: *nop, addi, movi, bgt, jd, jmp, ld, st.*
- States: each state is a tuple that includes
  - code memory $M_c$
  - data memory $M_d$
  - registers $R$
  - program counter $pc$
- Steps: transition relations define the possible state changes of the machine.
1. A small model of a machine

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nop w</code></td>
<td>`(M_c</td>
<td>M_d, R, pc + 1)<code>, when </code>pc + 1 ∈ dom(M_c)`</td>
</tr>
<tr>
<td><code>add r_d, r_s, r_t</code></td>
<td>`(M_c</td>
<td>M_d, R{r_d → R(r_s) + R(r_t)}, pc + 1)<code>, when </code>pc + 1 ∈ dom(M_c)`</td>
</tr>
<tr>
<td><code>addi r_d, r_s, w</code></td>
<td>`(M_c</td>
<td>M_d, R{r_d → R(r_s) + w}, pc + 1)<code>, when </code>pc + 1 ∈ dom(M_c)`</td>
</tr>
<tr>
<td><code>movi r_d, w</code></td>
<td>`(M_c</td>
<td>M_d, R{r_d → w}, pc + 1)<code>, when </code>pc + 1 ∈ dom(M_c)`</td>
</tr>
</tbody>
</table>
| `bgt r_s, r_t, w` | `(M_c | M_d, R, w)`, when `R(r_s) > R(r_t) ∧ w ∈ dom(M_c)`  
`(M_c | M_d, R, pc + 1)`, when `R(r_s) ≤ R(r_t) ∧ pc + 1 ∈ dom(M_c)` |
| `jd w` | `(M_c | M_d, R, w)`, when `w ∈ dom(M_c)` |
| `jmp r_s` | `(M_c | M_d, R, R(r_s))`, when `R(r_s) ∈ dom(M_c)` |
| `ld r_d, r_s(w)` | `(M_c | M_d, R{r_d → M(R(r_s) + w)}, pc + 1)`, when `pc + 1 ∈ dom(M_c)` |
| `st r_d(w), r_s` | `(M_c | M_d{R(r_d) + w → R(r_s)}, R, pc + 1)`, when `R(r_d) + w ∈ dom(M_d) ∧ pc + 1 ∈ dom(M_c)` |
1. A small model of a machine

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nop w</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>add $r_d, r_s, r_t$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>addi $r_d, r_s, w$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>movi $r_d, w$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>bgt $r_s, r_t, w$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>jdp w</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>jmp $r_s$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>ld $r_d, r_s(w)$</strong></td>
<td>$(M_c</td>
</tr>
<tr>
<td><strong>st $r_d(w), r_s$</strong></td>
<td>$(M_c</td>
</tr>
</tbody>
</table>

$Dc$ : instruction decoding function
# 1. A small model of a machine

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{nop} $w$</td>
<td>$(M_c</td>
</tr>
<tr>
<td>\textit{add} $r_d, r_s, r_t$</td>
<td>$(M_c</td>
</tr>
<tr>
<td>\textit{addi} $r_d, r_s, w$</td>
<td>$(M_c</td>
</tr>
<tr>
<td>\textit{movi} $r_d, w$</td>
<td>$(M_c</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{bgt } r_s, r_t, w & \\
\text{If } Dc(M_c(pc)) = \text{jmp } r_s & \quad R(r_s) \in \text{dom}(M_c) \\
& \quad (M_c|M_d, R, pc) \rightarrow_n (M_c|M_d, R, R(r_s)) \\
\text{jd } w & \\
\text{jmp } r_s & \quad (M_c|M_d, R, R(r_s)), \text{ when } R(r_s) \in \text{dom}(M_c) \\
\text{ld } r_d, r_s(w) & \quad (M_c|M_d, R\{r_d \mapsto M(R(r_s) + w)\}, pc + 1)$, when $pc + 1 \in \text{dom}(M_c)$ |
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{st} $r_d(w), r_s$</td>
<td>$(M_c</td>
</tr>
</tbody>
</table>
1. A small model of a machine

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nop w</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>add r_d, r_s, r_t</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>addi r_d, r_s, w</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>movi r_d, w</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>bgt r_s, r_t, w</code></td>
<td>(Dc(M_c(pc)) = jmp \ r_s \quad R(r_s) \in \text{dom}(M_c))</td>
</tr>
<tr>
<td></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>jd w</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>jmp r_s</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>ld r_d, r_s(w)</code></td>
<td>((M_c</td>
</tr>
<tr>
<td><code>st r_d(w), r_s</code></td>
<td>((M_c</td>
</tr>
</tbody>
</table>

+ \(M_d\) could change at any time (because of attacker actions).
2. Example condition on instrumentation

Computed jumps occur only in context of a specific instruction sequence:

```
addi r0, rs, 0
ld r1, r0(0)
mov r2, IMM
bgt r1, r2, HALT
bgt r2, r1, HALT
jmp r0
```
2. Example condition on instrumentation

Computed jumps occur only in context of a specific instruction sequence:

\[
\begin{align*}
\text{addi } r_{0}, r_{s}, 0 \\
\text{ld } r_{1}, r_{0}(0) \\
\text{movi } r_{2}, \text{IMM} \\
\text{bgt } r_{1}, r_{2}, \text{HALT} \\
\text{bgt } r_{2}, r_{1}, \text{HALT} \\
\text{jmp } r_{0}
\end{align*}
\]

\textit{HALT} is the address of a halt instruction.
\textit{IMM} is a constant that encodes the allowed label at the jump target.
3. A result

Let $S_0$ be a state with $\text{pc} = 0$ and code memory $\mathcal{M}_c$ that satisfies the instrumentation condition for a given CFG. Suppose $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$ where each $\rightarrow$ transition is either a normal $\rightarrow_n$ step or an attacker step that changes only data memory. For each $i$, if $S_i \rightarrow_n S_{i+1}$ then $\text{pc}$ at $S_{i+1}$ is one of the allowed successors of $\text{pc}$ at $S_i$ according to the CFG.
Software-based fault isolation

- CFI does not assume memory protection.
- But it enables memory protection, i.e., “software-based fault isolation” (SFI).
- Again, there are several possible implementations of SFI.
  - E.g., by code rewriting, with guards on memory operations.
- Recent systems (XFI, BGI, LXFI, NaCl, ...) explore several variants and extensions.
A recent system: Native Client (NaCl) [Yee et al.]
A recent SFI tool: RockSalt
[Morrisett et al.]

- RockSalt is an SFI checker
  - for the NaCl sandbox policy,
  - ~80 lines of Coq code, manually translated into C.

- A formal argument shows that, if RockSalt accepts a string of bytes B, then B’s execution on x86 will respect the sandbox policy.
  - The argument is based on a sophisticated Coq model of x86 integer instructions.
  - More work remains, in several directions: models, proofs, policies.
Some themes
Some themes

• Inventive attackers, with deep, detailed understanding of their targets.
Some themes

• Inventive attackers, with deep, detailed understanding of their targets.

• The malleability of software:
  – enables sophisticated architectures and methods for protection,
  – benefits from looseness in systems constraints ("our goal is not to preserve semantics, but to improve it"),
  – costs in compatibility and run-time efficiency.
Reading

• Aleph One’s “Smashing the stack for fun and profit”
  http://www.insecure.org/stf/smashstack.txt

• Pincus & Baker’s “Beyond stack smashing: Recent advances in exploiting buffer overruns”
  http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1324594&tag=1

• Erlingsson’s “Low-level Software Security: Attacks and Defenses”
Exercise 1:
In MicroIL, are the following two programs well-typed, with respect to any $F$ and $S$? (yes/no). If so, give one pair of suitable $F$ and $S$ (by defining $F_1$, $F_2$, $F_3$, $S_1$, $S_2$, and $S_3$.)

a) push0 · inc · halt

b) inc · inc · halt
Homework 4

Exercise 2:
Re. Kennedy’s Problem 4, sketch a small example of a function $g$ that illustrates the difficulty being discussed in Section 3 (p9).
Exercise 3:
Erlingsson’s paper describes six defense techniques (and some variants). Summarize which of them rely on the secrecy of certain information.