SOURCE-FREE BINARY SOFTWARE
SECURITY RETROFITTING

DR. KEVIN HAMLEN
LOUIS A. BEECHERL, JR. DISTINGUISHED PROFESSOR
COMPUTER SCIENCE DEPARTMENT
CYBER SECURITY RESEARCH AND EDUCATION INSTITUTE, EXECUTIVE DIRECTOR
THE UNIVERSITY OF TEXAS AT DALLAS

Supported in part by:
NSF Award #1513704 & NSF CAREER Award #1054629,
AFOSR YIP (Career) Award FA9550-08-1-0044,
AFOSR Active Defense Grant FA9550-10-1-0088,
AFOSR Award FA9550-14-1-0173,
ONR Awards N00014-14140030 & N00014-17-1-2995,
and an NSF I/UCRC Awards from Raytheon & Lockheed Martin

Any opinions, findings, conclusions, or recommendations expressed in this presentation are those of the author(s) and do not necessarily reflect the views of the AFOSR, NSF, ONR, Raytheon, or Lockheed Martin.
Mission-critical Software Environments

- **Myth:** In mission-critical environments, all software is custom, rigorously tested, and formally verified.

- **Reality:** Most mission-critical environments use commodity software and components extensively.
  - Commercial Off-The-Shelf (COTS)
    - widely available to attackers
  - mostly closed-source
    - independent security audit not feasible
  - supports mainstream OSes (Windows) and architectures (Intel)
  - some effort at secure development, but no formal guarantees
Critical Infrastructure: Critically Insecure

- 2010: Stuxnet infiltrates and destroys Iranian nuclear centrifuges
  - **Software exploited**: Siemens Windows apps and PLCs
  - Sets Iranian nuclear program back 3-5 years

- 2020: Hundreds of US infrastructure networks penetrated by SolarWinds hack
  - **Software exploited**: Microsoft Exchange
  - Supply-line hack infects network monitors at Pentagon, Treasury, Microsoft, Intel, Cisco, …

- 2021: Colonial Oil Pipeline Hack
  - **Software exploited**: Unpatched Windows VPN
  - Leaked password to unused account, no multifactor authentication, no data backups
  - weeks of oil shortages in eastern US, tens of thousands of miles of pipeline checks

- 2020: Hundreds of US infrastructure networks penetrated by SolarWinds hack
  - **Software exploited**: Microsoft Exchange
  - Supply-line hack infects network monitors at Pentagon, Treasury, Microsoft, Intel, Cisco, …
(In)famous Linux Vulnerabilities

- **Heartbleed**
  - OpenSSL vulnerability disclosed April 2014
  - allowed anyone to anonymously grab arbitrary data (e.g., master keys) from internet-facing services
  - affected ~66% of all web servers, email servers, chat servers, VPNs, clients, etc.
  - all versions vulnerable since 2011!

- **Shellshock**
  - Bash shell vulnerability disclosed September 2014
  - allowed complete compromise - remote code execution
  - all versions vulnerable since 1989(!!)
Are In-house Projects “More Secure”?

- **Idea:** Build all your own custom software in-house from scratch (or contract trusted third-party to build from scratch).
  - expensive, time-consuming
  - error-prone (not built by specialists)
    - 63% of in-house IT projects fail to meet their own specs [CHAOS Report]
  - poor compatibility, hard to maintain
  - very questionable security assurance
    - vulnerable to insider threats, less tested, shaky design, etc.
    - assurance usually based on myth of “security by obscurity”

- **Many COTS advantages**
  - constantly updated for new threats
  - tested on a mass scale
  - crafted & maintained by specialists
  - cheaper, mass-produced
Why is Software so Insecure?

- Huge and constantly evolving
  - Windows XP has 40 million lines of code
  - Microsoft Office had 30 million lines in 2006
  - Debian 5.0 has a staggering 500 million lines!
    - contrast: Space shuttle has only 2.5 million moving parts!

- Often written in unsafe languages
  - C, C++, VC++, Visual Basic, scripting languages, ...

- Increasingly sophisticated attacks
  - buffer-overrun
  - direct code-injection
  - return-to-libc
  - return-oriented programming (RoP)
  - implementation disclosure-assisted code-reuse attacks
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    …
    return;
}
```

```
lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data "*.*"
.data "aaaaa…"
.data "aaaa"
<addr of buf>
```

```
8D 45 B8 50 FF 15 BC 82 2F 01
65 72 61 73 65 20 61 61 61 61
30 FB 1F 00
```

```
65 72 61 73 65 20 2A 2E 2A 01
62 61 61 61 30 FB 1F 00
```

```
leaves space for:
- argc (4 bytes)
- argv (4 bytes)
- buf (64 bytes)
- saved EBP (4 bytes)
- saved EIP (4 bytes)
- argv (4 bytes)
- argc (4 bytes)
- bottom of stack (higher addresses)
- top of stack (lower addresses)
```
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

![Illustration of code-injection example with assembly code and data segment](image)
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

lea eax,[ebp-48h]
push eax
call <system>
data "erase"
data "*. *
.data "aaaaa...
data "aaaa"
<addr of buf>

lea eax,[ebp-48h]
push eax
call <system>
erase *. * aaaaaaaaaaaaaaaaaaaaaaaaa
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

debugger output:
lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data ".*"
data "aaaaaaa..."
data "aaaa"
<addr of buf>

stack layout:
bottom of stack (higher addresses)
argc (4 bytes)
argv (4 bytes)
<addr of buf>

Top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaa aaaaaaaaaaaaaaaaa
 aaaa
Code-injection Example

```c
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```

```
8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

```
top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>

bottom of stack (higher addresses)
```
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ... return;
}
```

8D 45 B8  lea eax,[ebp-48h]
50          push eax
FF 15 BC 82 2F 01  call <system>
65 72 61 73 65 20 .data "erase"
2A 2E 2A 20    .data "*.*"
61 (x24)      .data "aaaaa..."
61 61 61 61    .data "aaaa"
30 FB 1F 00   <addr of buf>

top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaaaaaaaaaa
aaaaa
<addr of buf>
argv (4 bytes)
<addr of "erase *.* ...">
bottom of stack (higher addresses)
Pernicious Vulnerabilities
[SourceFire Vulnerability Research]

TOP HIGH SEVERITY VULNERABILITIES

- Buffer Errors: 24%
- SQL Injection: 21%
- Code Injection: 10%
- Access Control: 10%
- Not enough info: 8%
- Input Validation: 7%
- Resource Management: 4%
- Path Traversal: 3%
- Everything Else: 13%
Defense: DEP + ASLR

- Data Execution Prevention (DEP)
  - set stack memory non-executable (hardware-enforced)

- Address Space Layout Randomization (ASLR)
  - randomize locations of libraries on-load

- Counter-attack
  - don’t insert any code onto the stack
  - jump directly to existing code fragments
  - called a “code-reuse” attack
ROP Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

<table>
<thead>
<tr>
<th>Stack Frame Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller’s stack frame</td>
</tr>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
<tr>
<td>saved EBP (4 bytes)</td>
</tr>
<tr>
<td>saved EIP (4 bytes)</td>
</tr>
<tr>
<td>top of stack (lower addresses)</td>
</tr>
<tr>
<td>buf (64 bytes)</td>
</tr>
<tr>
<td>bottom of stack (higher addresses)</td>
</tr>
</tbody>
</table>
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

.data "erase"
.data "*.*"
.data "aaaa..."
.data <addr1>
.data <addr2>
.data <addr2>
.data <addr3>

top of stack (lower addresses)
    erase *.*
    aaaaaaaaa...
ROP Example

init_display: ...
   < ... 1024 bytes ... >

system: ...

addr2: add eax, 512
    ret
    ...

addr1: mov eax, [init_display]
    call eax
    pop ebx
    ret
    ...

addr3: call eax
    ret
ROP Example

init_display: ...
< ... 1024 bytes ... >

system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display

top of stack (lower addresses)

erase *.*
aaaaaaa...

<addr1>

<addr2>

<addr2>

<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display

top of stack (lower addresses)
erase *.*
aaaaaaa...

aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr1: mov eax, [init_display]
call eax
pop ebx
ret

addr2: add eax, 512
ret

addr3: call eax
ret

top of stack (lower addresses)
 erase *.*
  aaaaaaa...

  aaaa
  <addr1+5>
  aaaa
  <addr2>
  <addr2>
  <addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >

system: ...

addr2: add eax, 512
ret
...

addr1: mov eax, [init_display]
call eax
pop ebx
ret
...

addr3: call eax
ret

eax = init_display

top of stack (lower addresses)
erase *.*

aaaaaaa...

aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2:  
  add eax, 512
  ret
  ...

addr1:  
  mov eax, [init_display]
  call eax
  pop ebx
  ret
  ...

addr3:  
  call eax
  ret

eax = init_display

top of stack (lower addresses)

erase *.*
aaaaaaa...

< ... 1024 bytes ... >
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display+512

top of stack (lower addresses)
erase *.*
aaaaaaaa...

< ... 1024 bytes ... >
system: ...

eax = init_display+512

top of stack (lower addresses)
erase *.*

aaaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
call eax
pop ebx
ret
...
addr3: call eax
ret
eax = init_display+512

top of stack (lower addresses)
erase *.*
aaaaaaaa...

aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >

system: ...

addr1: mov eax, [init_display]
call eax
pop ebx
ret

addr2: add eax, 512
ret
...
addr3: call eax
ret

eax = init_display+1024 = system !!!

top of stack (lower addresses)
erase *. *

aaaaaaa...

<addr1+5>

aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display+1024 = system !!!
top of stack (lower addresses)
erase *.*
aaaaaaaa...

aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2:
   add eax, 512
   ret
   ...

addr1:
   mov eax, [init_display]
   call eax
   pop ebx
   ret
   ...

addr3:
   call eax
   ret

eax = init_display + 1024 = system !!!

top of stack (lower addresses)

erase *.*

aaaaaaa...

aaaa

<addr1+5>

aaaa

<addr2>

<addr2>

<addr3>
Battling Code-reuse Attacks

- **Microsoft’s 2012 BlueHat Competition**
  - Focused on RoP Mitigation
  - $260,000 total for top three solutions
    - Successful attack against 2nd place solution was published two weeks later

- **Google Pwnium Competition**
  - Hacker Pinkie Pie paid $60K for Chrome RoP exploit
  - Google fixes the exploit
  - Five months later, Pinkie Pie finds a new RoP exploit in the fixed Chrome, gets paid another $60K
  - Google fixes the 2nd exploit
  - Five months later, Pinkie Pie finds a yet another (partial) exploit, gets paid another $40K
Code-reuse Conflict Timeline

- Stack smashing
- Return into libc
- ASLR bypass
- Borrowed code
- ROP
- ROP on ARM & SPARC
- JIT spraying
- ROP w/o returns
- JOP
- JIT ROP
- Side-channel attacks on div. code
- Blind ROP
- Gadget-stitching
- JIT ROP++
- JIT spray on ARM

- Stackguard
- Stack layout randomization
- ASLR
- DEP
- CFI
- ASLP
- Shadow stacks
- Randomization
- Heap randomization
- C-FREE
- librado
- Bin-CFI
- kBouncer
- SafeDispatch
- Oxymoron
- XnR Memory
- Modular CFI & CFI for JITs
- CPI
- Forward-Edge CFI
My Research: Security Retrofitting

Secure commodity software AFTER it is compiled and distributed, by automatically modifying it at the binary level.
Advantages

- No need to get code-producer cooperation
- No need to customize the OS/VM
- No custom hardware needed (expensive & slow)
- Not limited to any particular source language or tool chain
- Can enforce consumer-specific policies
- Maintainable across version updates (just re-apply rewriter to newly released version)
- Rewriter remains untrusted, so can outsource that task to an untrusted third party!
  - Local, trusted verifier checks results
Challenges

- Software is in purely binary form
  - no source, no debug info, no disassembly

- Diverse origins
  - various source languages, compilers, tools, ...

- Code-producers are uncooperative
  - unwilling to recompile with special compiler
  - unwilling to add/remove features
  - no compliance with any coding standard

- Highly complex binary structure
  - target real-world APIs (e.g., hundreds of thousands of Windows system dll’s and drivers)
  - multi-threaded, multi-process
  - event-driven (callbacks), dynamically linked (runtime loading)
  - heavily optimized (binary code & data arbitrarily interleaved)
Three Major Advances

1) Heuristic-free & Machine Learning-based Binary Disassembly
   - automatically recovers high-level program structure from binary software product
   - Superset Disassembly (NDSS’18): recover a superset of the control-flow graph
   - Finding the Undecidable Path (PAKDD’14): Optimize CFG via machine learning

2) Native Code Instrumentation
   - method of automatically in-lining extra security checks into untrusted programs

3) Formal, Automated, Machine-validation
   - automatically PROVES (mathematically) that retrofitted software is immune to certain classes of attacks
First Step: Disassembly

- Disassemble this hex sequence
- Turns out x86 disassembly is an undecidable problem!

<table>
<thead>
<tr>
<th>Valid Disassembly</th>
<th>Valid Disassembly</th>
<th>Valid Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E0</td>
<td>jmp eax</td>
<td>FF E0</td>
</tr>
<tr>
<td>5B</td>
<td>pop ebx</td>
<td>5B</td>
</tr>
<tr>
<td>5D</td>
<td>pop ebp</td>
<td>5D</td>
</tr>
<tr>
<td>C3</td>
<td>retn</td>
<td>C3</td>
</tr>
<tr>
<td>0F 88 52</td>
<td>jcc</td>
<td>0F 88</td>
</tr>
<tr>
<td>0F 84 EC</td>
<td></td>
<td>0F 84 EC</td>
</tr>
<tr>
<td>8B ...</td>
<td>mov</td>
<td>88 52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0F 84 EC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8B ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disassembly Intractability

- Even the best reverse-engineering tools cannot reliably disassemble even standard COTS products
- Example: IDA Professional Disassembler (Hex-rays)

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Disassembly Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Foundation Class Lib (mfc42.dll)</td>
<td>1216</td>
</tr>
<tr>
<td>Media Player (mplayerc.exe)</td>
<td>474</td>
</tr>
<tr>
<td>Avant Web Browser (RevelationClient.exe)</td>
<td>36</td>
</tr>
<tr>
<td>VMWare (vmware.exe)</td>
<td>183</td>
</tr>
</tbody>
</table>
Innovation: Superset Disassembly

Byte Sequence: FF E0 5B 5D C3 0F 88 B0 50 FF FF 8B

Hex
- FF
- E0
- 5B
- 5D
- C3
- 0F
- 88
- B0
- 50
- FF
- FF
- 8B

Disassembled

Invalid

Included Disassembly
- jmp eax
- pop
- L1: pop
- retn
- jcc
- L2: mov
- loopne
- jmp L1
- mov
- jmp L2
Problem: Pointers

- We just rearranged everything. Pointers will all point to the wrong places.
  - can’t reliably identify pointer data in a sea of unlabeled bytes
- Two kinds of relevant pointers:
  - pointers to static data bytes among the code bytes
  - pointers to code (e.g., method dispatch tables)
Preserving Static Data Pointers

- Put the de-shingled code in a NEW code segment.
  - Set it execute-only (non-writable)
- Leave the original .text section
  - Set it read/write-only (non-execute)

Original Binary

| Header | Import Address Table | .data | .text |

Rewritten Binary

| Header | Import Address Table | .data | .told (NX bit set) |

| .tnew (de-shingled code) |
Preserving Code Pointers

- Almost half of all jump instructions in real x86 binaries compute their destinations at runtime.
  - Exercise: Why? Examples?
  - ...
  - ...
  - ...
  - ...

- Must ensure these jumps target new code locations instead of old.
  - impossible to statically predict their destinations
Almost half of all jump instructions in real x86 binaries compute their destinations at runtime.

- all method calls (read method dispatch table)
- all function returns (read stack)
- almost all API calls (read linker tables)
- pointer encryption/decryption logic for security

Must ensure these jumps target new code locations instead of old.

- impossible to statically predict their destinations
Solution: Control-flow Patching

- Create a lookup table that maps old code addresses to new ones at runtime.
- Add instructions that consult the lookup table before any computed jump.

<table>
<thead>
<tr>
<th>Original</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump eax</td>
<td>jump table[eax]</td>
</tr>
</tbody>
</table>
Optimizing

- With these three tricks we can successfully transform (most) real-world COTS binaries even without knowing how they work or what they do!
  - de-shingling disassembly
  - static data preservation
  - control-flow patching

- Limitations
  - runtime code modification conservatively disallowed
  - computing data pointers from code pointers breaks
  - These are compatibility limitations not security limitations.

- But it’s prohibitively inefficient (increases code size ~700%)
  - need to optimize the approach
Optimization Philosophy

1. If the optimization fails, we might get broken code but never unsafe code.

2. The optimizations only need to work for non-malicious, non-vulnerable code fragments.
   - If the code fragment is malicious or vulnerable, we don’t want to preserve it!
**Optimization #1: De-shingling**

- Lots of extra overlapping information
  - Can we prune our disassembly tree?

<table>
<thead>
<tr>
<th>Hex</th>
<th>Path 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>jmp eax</td>
</tr>
<tr>
<td>0F</td>
<td>retn</td>
</tr>
<tr>
<td>B0</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td></td>
</tr>
<tr>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>pop</td>
</tr>
<tr>
<td>5B</td>
<td>L1: pop</td>
</tr>
<tr>
<td>5D</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>L2: mov</td>
</tr>
<tr>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>E0</td>
<td>pop</td>
</tr>
<tr>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>
Machine learning-based Disassembler

- Insight: Distinguishing real code bytes from data bytes is a “noisy word segmentation problem”.
  - Word segmentation: Given a stream of symbols, partition them into words that are contextually sensible. [Teahan, 2000]
  - Noisy word segmentation: Some symbols are noise (data).

- Machine Learning based disassembler
  - based on $k$th-order Markov model
  - Estimate the probability of the sequence $B$:

\[
p(B|M_\alpha) = - \log \prod_{i=1}^{\lvert B \rvert} p(b_i|b_{i-k}^{i-1}, M_\alpha)
\]


Disassembler Stats

# of instructions identified by our disassembler but not by IDA Pro
## PPM Disassembly Stats

<table>
<thead>
<tr>
<th>PPM Disassembler</th>
<th>False Negative</th>
<th>False Positive</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>7zFM</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>notepad</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>DosBox</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>WinRAR</td>
<td>0</td>
<td>39</td>
<td>99.982%</td>
</tr>
<tr>
<td>mulberry</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>scummvm</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>emule</td>
<td>0</td>
<td>117</td>
<td>99.988%</td>
</tr>
<tr>
<td>Mfc42</td>
<td>0</td>
<td>47</td>
<td>99.987%</td>
</tr>
<tr>
<td>mplayerc</td>
<td>0</td>
<td>307</td>
<td>99.963%</td>
</tr>
<tr>
<td>revClient</td>
<td>0</td>
<td>71</td>
<td>99.893%</td>
</tr>
<tr>
<td>vmware</td>
<td>0</td>
<td>45</td>
<td>99.988%</td>
</tr>
</tbody>
</table>
Optimization #2: Lookup Table Compression

- **Idea:** Overwrite the old code bytes with the lookup table.
  - PPM disassembler identifies most code bytes
  - Also identifies subset that are possible computed jump destinations.
  - Overwrite those destinations with our lookup table.

<table>
<thead>
<tr>
<th>Original</th>
<th>Rewritten</th>
</tr>
</thead>
</table>
| call eax | cmp [eax], 0xF4  
cmovz eax, [eax+1]  
call eax |
Applications of our Rewriter

Three Applications

- Binary randomization for RoP Defense (STIR)
- Opaque Control-Flow Integrity (O-CFI)
- Machine-certified Software Fault Isolation (Reins)
RoP Defense Strategy

- RoP is one example of a broad class of attacks that require attackers to know or predict the location of binary features

Defense Goal
Frustrate such attacks by randomizing the feature space
- Randomly reorder the program’s internal layout every time the program loads
  - Attacker cannot reliably locate code addresses for code-reuse attacks
  - Astronomically low chance of attack success
  - Exact attack probability is \textit{mathematically computable} as an entropy calculation
Supports Windows PE and Linux ELF files

Tested on SPEC2000 benchmarks and the entire coreutils chain for Linux

1.5% program runtime efficiency overhead on average


- Won 2nd place in the NYU-Poly AT&T Best Applied Security Paper of the Year competition


- Conceals code reachability info to defeat even advanced attackers who can inspect portions of the randomized program memory image!
Gadget Reduction

<table>
<thead>
<tr>
<th>Application</th>
<th>Gadget Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dosbox</td>
<td>99.99%</td>
</tr>
<tr>
<td>Notepad++</td>
<td>99.99%</td>
</tr>
<tr>
<td>gzip</td>
<td>99.99%</td>
</tr>
<tr>
<td>vpr</td>
<td>99.99%</td>
</tr>
<tr>
<td>mdf</td>
<td>99.99%</td>
</tr>
<tr>
<td>parser</td>
<td>99.99%</td>
</tr>
<tr>
<td>gap</td>
<td>99.99%</td>
</tr>
<tr>
<td>bzip2</td>
<td>99.99%</td>
</tr>
<tr>
<td>twolf</td>
<td>99.99%</td>
</tr>
<tr>
<td>mesa</td>
<td>99.99%</td>
</tr>
<tr>
<td>art</td>
<td>99.99%</td>
</tr>
<tr>
<td>quake</td>
<td>99.99%</td>
</tr>
</tbody>
</table>
Windows STIR Runtime Overhead

- gzip
- vpr
- mcf
- parser
- gap
- bzip2
- twolf
- mesa
- art
- equake
Custom Safety Policy Enforcement with Machine-provable Assurance

- **Untrusted Binary Code**
- **Binary Rewriter**
  - Safety Policy
  - Secure Binary
- **Verifier**
  - Deploy
  - Reject

Diagram: Flowchart showing the process of custom safety policy enforcement with machine-provable assurance.
An API Policy

function conn = ws2_32::connect(
    SOCKET, struct sockaddr_in *, int) -> int;

function cfile = kernel32::CreateFileW(
    LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
    DWORD, DWORD, HANDLE) -> HANDLE WINAPI;

event e1 = conn(_, {sin_port=25}, _) -> 0;

event e2 = cfile("*.exe", _, _, _, _, _, _) -> _;

policy = e1* + e2*;

Policy: Applications may not both open email connections and create files whose names end in "exe".
Reference Monitor In-lining

- In-line security checks as rewriting progresses
  - checks uncircumventable due to control-flow and memory safety
  - ensures complete mediation
REINS - Rewriting and In-lining System

- Prototype targets full Windows XP/7/8 OS
  - significantly harder than Linux
- 2.4% average runtime overhead
- 15% average process size increase
- Tested on SPEC2000, malware, and large GUI binaries
  - Eureka email client and DOSBox, much larger than any previous implementation had accomplished
  - won Best Student Paper at ACSAC
Control-Flow Safety

- Used PittSField approach [McCamant & Morrisett, 2006]
  - Break binaries into chunks
    - chunk – fixed length (16 byte) basic blocks
  - Only one extra guard instruction necessary
  - Mask instruction only affects violating flows

<table>
<thead>
<tr>
<th>Original</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>call eax</td>
<td>cmp [eax], 0xF4</td>
</tr>
<tr>
<td></td>
<td>cmovz eax, [eax+1]</td>
</tr>
<tr>
<td></td>
<td>and eax, 0x0FFFFFFF0</td>
</tr>
<tr>
<td></td>
<td>call eax</td>
</tr>
</tbody>
</table>
## Jump Table w/ Masking

### Original Instruction:

| .text: 0040CC9B | FF DO | call eax |

### Original Possible Target:

| .text: 00411A40 | 5B | pop ebp |

### Rewritten Instructions:

| .tnew: 0052A1C0 | 80 38 F4 | cmp byte ptr [eax], F4h |
| .tnew: 0052A1C3 | 0F 44 40 01 | cmovz eax, [eax+1] |
| .tnew: 0052A1C7 | 80 38 F4 | and eax, 0x0FFFFFF0 |
| .tnew: 0052A1CE | FF D0 | call eax |

### Rewritten Jump Table:

| .told: 00411A40 | F4 B9 4A 53 00 | F4 dw 0x534AB0 |

### Rewritten Target:

| .tnew: 00534AB0 | 5B | pop ebp |
Next Two Lectures

- **Wednesday:** Some of our most recent work for Navy and DARPA
  - automated binary software *attack surface reduction* using technologies underlying STIR

- **Monday:** The sciences behind it all...
  - Theory of In-lined Reference Monitors (IRMs)
  - Computability theory and Enforceability theory


