CE 874 - Secure Software Systems

Reassembly

Mehdi Kharrazi  
Department of Computer Engineering  
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
Run-Time protection/enforcement

• In many instances we only have access to the binary
• How do we analyze the binary for vulnerabilities?
• How do we protect the binary from exploitation?
• This would be our topic for the next few lectures
Why Binary Code?

• Access to the source code often is not possible:
  • Proprietary software packages
  • Stripped executables
  • Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
• Binary code is the only authoritative version of the program
  • Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary
• Worms and viruses are rarely provided with source code
Goals for the day

- Last time we discussed binary analysis
  - Binary Analysis
  - Binary patching/rewriting
  - Binary instrumentation
    - Very short discussion of CFI
    - Taint analysis
- Today we want to discuss:
  - another use case for binary patching
  - why reassembly (i.e. binary re-writing) is hard?
Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code
R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. CCS 2012
Attacks Timeline

1980: Execute Code on the Stack
1990: Return to Unsafe Library (return-to-libc)
2000: Return to Unsafe User Code Gadgets (Shacham, Q [8,1])
2010: Make Stack Non-exec (WxorX)

Randomize Library Image Base (ASLR)

? [Wartell’12]
RoP Attack

Attacker Smashes the Stack!
RoP Attack

```
0x6D78941C: retn

Gadg1: 0x6D8011AC: add esp, 12
0x6D8011AF: retn

Gadg4: 0x6D802A88: pop edi
0x6D802A89: retn

Gadg3: 0x6D81BDD7: pop ecx
0x6D81BDD8: retn

Gadg2: 0x6D8FF626: mov eax, ebx
0x6D8FF628: pop ebx
0x6D8FF629: pop ebp
0x6D8FF62A: retn

Gadg5: 0x6D97ED06: sub ecx, edx
0x6D97ED08: push edi
0x6D97ED09: push ecx
0x6D97ED0A: call [IAT:X]
```

**Attack Success!**

**Action:** Store `<var_5>` in edi for later use
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 feature space or removing features
RoP Defenses: Compiler-based

- Control the machine code instructions used in compilation (Gfree [2] and Returnless [3])
  - Use no return instructions
  - Avoid gadget opcodes
- Hardens against RoP
- Requires code producer cooperation
  - Legacy binaries unsupported

```
let rec merge = function
  | list, []        -> list
  | [], list -> list
  | h1::t1, h2::t2 ->
    if h1 <= h2 then
      h1 :: merge (t1, h2::t2)
    else
      h2 :: merge (h1::t1, t2);
```

Gadget-removing Compiler

Gadget-free Binary
GFree Alignment Sled
RoP Defenses: ASLR

- ASLR randomizes the image base of each library
  - Gadgets hard to predict
  - Brute force attacks still possible [4]
RoP Defenses: IPR / ILR

- Instruction Location Randomization (ILR) [5]
  - Randomize each instruction address using a virtual machine
  - Increases search space
  - Cannot randomize all instructions
  - High overhead due to VM (13%)

- In-place Randomization (IPR) [6]
  - Modify assembly to break known gadgets
  - Breaks 80% of gadgets on average
  - Cannot remove all gadgets
  - Preserves gadget semantics
  - Deployment issues
Our Goal

• Self-randomizing COTS binary w/o source code
  • Low runtime overhead
  • Complete gadget removal
  • Flexible deployment (copies randomize themselves)
  • No code producer cooperation
Challenge: Binary Randomization w/o metadata

- Relocation information, debug tables and symbol stores not always available
  - Reverse engineering concerns
- Perfect static disassembly without metadata is provably undecidable
  - Best disassemblers make mistakes (IDA Pro)

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction Count</th>
<th>IDA Pro Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfc42.dll</td>
<td>355906</td>
<td>1216</td>
</tr>
<tr>
<td>mplayerc.exe</td>
<td>830407</td>
<td>474</td>
</tr>
<tr>
<td>vmware.exe</td>
<td>364421</td>
<td>183</td>
</tr>
</tbody>
</table>
Unaligned Instructions

- Disassemble this hex sequence
  - Undecidable problem

```
FF E0 5B 5D C3 0F
88 52 0F 84 EC 8B
```

<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</td>
</tr>
<tr>
<td>0F 84 EC</td>
<td>mov</td>
<td>88 52</td>
</tr>
<tr>
<td>8B ...</td>
<td>mov</td>
<td>84 EC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8B ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8B ...</td>
</tr>
</tbody>
</table>

[Wartell’12]
Our Solution: STIR (Self-Transforming Instruction Relocation)

- Statically rewrite legacy binaries to re-randomize at load-time
  - Greatly increases search space against brute force attacks
  - Introduces no deployment issues
  - Tested on 100+ Windows and Linux binaries
- 99.99% gadget reduction on average
- 1.6% overhead on average
- 37% process size increase on average
STIR Architecture

Original Application Binary → Binary Rewriter
Conservative Disassembler (IDA Python) → Lookup Table Generator → Self-stirring Binary → Memory Image
Load-time Randomizer (Helper Library) → Randomized Instruction Addresses

Static Rewriting Phase
Load-time Stirring Phase

[Wartell’12]
Static Rewriting

<table>
<thead>
<tr>
<th>Original Binary</th>
<th>Rewritten Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header</strong></td>
<td><strong>Rewritten Header</strong></td>
</tr>
<tr>
<td><strong>Import Address Table</strong></td>
<td><strong>Import Address Table</strong></td>
</tr>
<tr>
<td>.data</td>
<td>.data</td>
</tr>
<tr>
<td>.text</td>
<td>.told (NX bit set)</td>
</tr>
<tr>
<td>Block 1 -&gt; 500F86...</td>
<td>Block 1 -&gt; F4 &lt;NB 1&gt;</td>
</tr>
<tr>
<td>data -&gt; (8 bytes)</td>
<td>data -&gt; (8 bytes)</td>
</tr>
<tr>
<td>Block 2 -&gt; 55FF24...</td>
<td>Block 2 -&gt; F4 &lt;NB 2&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**.tnew**

| Denotes a section that is modified during static rewriting |

[Wartell’12]
Load-time Stirring

- When binary is loaded:
  - Initializer randomizes .tnew layout
  - Lookup table pointers are updated
  - Execution is passed to the new start address
Computed Jump Preservation

**Original Instruction:**
```
.text:0040CC9B | FF DO | call eax
```

**Original Possible Target:**
```
.text:0041A40 | 5B | pop ebp
```

**Rewritten Instructions:**
```
.tnew:0052A1CB | 80 38 F4 | cmp byte ptr [eax], F4h
tnew:0052A1CE | 0F 44 40 01 | cmovz eax, [eax+1]
tnew:0052A1D2 | FF D0 | call eax
```

**Rewritten Jump Table:**
```
.told:0041A40 | F4 B9 4A 53 00 | F4 dw 0x534AB9
```

**Rewritten Target:**
```
.tnew:00534AB9 | 5B | pop ebp
```

\[\text{Wartell'12}\]
Entropy Discussion

- ASLR
  - $2^{n-1}$ probes where $n$ is the number of bits of randomness

- STIR
  - $(2^n)! / (2(2^n - g)!)$ probes where $g$ is the number of gadgets in the payload
    - Must guess each where each gadget is with each probe.

- On a 64-bit architecture, the expected number of probes for a $g=3$-gadget attack is therefore over $7.92 \times 10^{28}$ times greater with STIR than with re-randomizing ASLR.

[Wartell’12]
Gadget Reduction

% of Gadgets Eliminated

Dosbox 99.92%
Notepad++ 99.92%
gzip 99.96%
vpr 99.98%
mdf 100.00%
parser 100.00%
gap 100.00%
bzip2 100.00%
twolf 100.00%
mesa 100.00%
art 100.00%
equake 100.00%

[Wartell’12]
Windows Runtime Overhead

SPEC2000 Windows Runtime Overhead

-9%  -5%  0%  5%  9%  14%  18%
gzip  vpr  mcf  parser  gap  bzip2  twolf  mesa  art  equake

[Wartell'12]
Linux Runtime Overhead

-15%  5%

base64  cat  cksum  comm  cp  expand  factor  fold  head  join  ls  md5sum  nl  od  paste  sha1sum  sha224sum  sha256sum  sha384sum  sha512sum  shred  shuf  unexpand  wc

Fall 1399  Ce 874 - Reassembly  [Wartell’12]
Conclusions

• First static rewriter to protect against RoP attacks
  • Greatly increases search space
  • Introduces no deployment issues
  • Tested on 100+ Windows and Linux binaries
  • 99.99% gadget reduction on average
  • 1.6% overhead on average
  • 37% process size increase on average
Problems with Binary Stirring

- Binary Stirring employs heuristics, which work on simple binaries
- Dynamic libraries are not considered in the evaluation
  - hence symbolization problem not addressed
Reassemblable Disassembling
Shuai Wang, Pei Wang, and Dinghao Wu, Usenix Security 2015
Motivation

• Analyzing and retrofiting COTS binaries with:
  • software fault isolation
  • control-flow integrity
  • symbolic taint analysis
  • elimination of ROP gadgets
• Binary rewriting comes with major drawbacks/limitations
  • runtime overhead from patching due to control-flow transfers
  • patching requires PIC if code is relocated
  • instrumentation significantly increases binary size
  • binary reuse only works for small binaries (coverage)
Goal

Produce reassembleable assembly code from stripped COTS binaries in a fully automated manner.

- Allows binary-based whole program transformations
- Requires relocatable assembly code $\rightarrow$ symbolization of immediate values
- Complementary to existing work
Symbolization

Given an immediate value in assembly code, is it a constant or a memory address?

• Reassembling transformed program changes binary layout
• Address changes invalidate memory references
• x86
  • No distinction between code and data
  • Variable-length instruction encoding
(Un)Relocatable Assembly Code

```
.text
mov 0xc0, %eax

.data
.long 0xa08

0xc0: 0xa08

mov 0xc0, %eax
asm
0xc0: ?

.text
mov Glob, %eax

.data
Glob:
.long 0xa08

mov Glob, %eax
asm
Glob: 0xa08
```
Disassemble

<table>
<thead>
<tr>
<th>.text</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>400100 mov [6000a0], eax</td>
<td></td>
</tr>
<tr>
<td>400105 jmp 0x40020d</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>40020d mov [6000a4], 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.data</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000a0 .long 0xc0debeef</td>
</tr>
<tr>
<td>6000a4 .long 0x0</td>
</tr>
</tbody>
</table>

[Fish’17]
...target

target:    mov [data_1], 1

.data

data_0 .long 0xc0debeef

data_1 .long 0x0

Disassemble
### Non-relocatable Assembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000a0</td>
<td>&quot;cat\x00&quot;</td>
</tr>
<tr>
<td>6000a4</td>
<td>.long 0x0</td>
</tr>
<tr>
<td>6000a8</td>
<td></td>
</tr>
</tbody>
</table>

**Disassemble**

Patch & Assemble

---

**Fall 1399**  
Ce 874 - Reassembly  
[Fish’17]
Disassemble Patch & Assemble

Relocatable Assembly

```assembly
.text
  mov [data_0], eax
  jmp target...
  ...
  mov [data_1], 1

.data
  new "cat\x00"
  data_0 .long 0xc0debeef
  data_1 .long 0x0
  data_1 .long 0xc0debeef

[CRASH!]
target
  mov [data_1], 1
```

[Fish’17]
Types of Symbol References

**Code Section**

```
fun1:
call fun2
        c2c

fun2:
    mov ptr, %eax
    lea (%eax, %ebx, 4), %ecx
    call *%ecx

handler1:
    ...

handler2:
    ...
```

**Data Section**

```
ptr:
    .long table
d2d
table:
    .long handler1
    .long handler2
c2d
```

[Wang’15]
Symbolization of c2c and c2d References

- Valid memory references point into code or data section
- Assume all immediates to be references and filter out invalid ones
Symbolization of d2c and d2d References

- Assumption 1
  - “All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.”
  - → Consider only n-byte values which are n-byte aligned

- Assumption 2
  - “Users do not need to perform transformation on the original binary data.”
  - → Keep start addresses of data sections during reassembly and ignore d2d references

- Assumption 3
  - “d2c symbol references are only used as function pointers or jump table entries.”
  - → References need to point to start of a function or form a jump table
Evaluation

- Uroboros: 13,209 SLOC in OCaml and Python; works with x86/x64 ELF binaries
- Intel Core i7-3770 @ 3.4GHz with 8GiB RAM running Ubuntu 12.04
- 122 programs compiled for 32- and 64-bit targets
- gcc 4.6.3 with default configuration and optimization of each program
- stripped before testing

<table>
<thead>
<tr>
<th>Collection</th>
<th>Size</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>COREUTILS</td>
<td>103</td>
<td>GNU Core Utilities</td>
</tr>
<tr>
<td>REAL</td>
<td>7</td>
<td>bc, ctags, gzip, mongoose, nweb, oftpd, thttpd</td>
</tr>
<tr>
<td>SPEC</td>
<td>12</td>
<td>C programs in SPEC2006</td>
</tr>
</tbody>
</table>
Architecture of Uroboros

Diagram:

- Binary
- Disassembly Module
  - Linear Disassembler
  - Disassembly Validator
- Data
- Meta-Data
- Code
- Analysis Module
  - Symbol Lifting
  - Control-Flow Structure Recovery
- Relocatable Assembly
  - External Analyses & Transformations

[Wang’15]
Correctness

- Test input shipped with programs or custom test of major functionality (some of REAL)

<table>
<thead>
<tr>
<th>Assumption Set</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>h264ref, gcc, gobmk, hmmer</td>
<td>perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lpm, sphinx3</td>
</tr>
<tr>
<td>{A1}</td>
<td>h264ref, gcc, gobmk</td>
<td>perlbench, gcc, gobmk</td>
</tr>
<tr>
<td>{A1, A2}</td>
<td>h264ref, gcc, gobmk</td>
<td>perlbench, gcc, gobmk</td>
</tr>
<tr>
<td>{A1, A3}</td>
<td>gobmk</td>
<td>gcc, gobmk</td>
</tr>
<tr>
<td>{A1, A2, A3}</td>
<td>gobmk</td>
<td>gobmk</td>
</tr>
</tbody>
</table>
Symbolization Errors

Table 4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Ref.</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FP Rate</td>
<td>FP</td>
<td>FP Rate</td>
<td>FP</td>
</tr>
<tr>
<td>perlbench</td>
<td>76538</td>
<td>2</td>
<td>0.026%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>hmmmer</td>
<td>13127</td>
<td>12</td>
<td>0.914%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>20600</td>
<td>27</td>
<td>1.311%</td>
<td>1</td>
<td>0.049%</td>
<td>1</td>
</tr>
<tr>
<td>gcc</td>
<td>262698</td>
<td>49</td>
<td>0.187%</td>
<td>32</td>
<td>0.122%</td>
<td>32</td>
</tr>
<tr>
<td>gobmk</td>
<td>65244</td>
<td>1348</td>
<td>20.661%</td>
<td>985</td>
<td>15.097%</td>
<td>912</td>
</tr>
</tbody>
</table>

Table 5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Ref.</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
<th>{}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FN</td>
<td>FN Rate</td>
<td>FN</td>
<td>FN Rate</td>
<td>FN</td>
</tr>
<tr>
<td>perlbench</td>
<td>76538</td>
<td>2</td>
<td>0.026%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>hmmmer</td>
<td>13127</td>
<td>12</td>
<td>0.914%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>20600</td>
<td>27</td>
<td>1.311%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>262698</td>
<td>11</td>
<td>0.042%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>gobmk</td>
<td>65244</td>
<td>86</td>
<td>1.318%</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
</tbody>
</table>
Overhead for REAL and SPEC

- No increase in binary size after first disassemble-assemble cycle

[Wang’15]
Conclusion

- Heuristic-based symbolization of memory references
- Uroboros provides re-assembleable disassembly
  - Available at https://github.com/s3team/uroboros
- Assumes availability of raw disassembly and function starting addresses
- Tested with gcc and Clang compiled binaries
- Limited support for C++ (need to parse DWARF)
Ramblr: Making Reassembly Great Again
Ruoyu “Fish” Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna, NDSS 2017
Disassemble

<table>
<thead>
<tr>
<th>.text</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>400100</td>
<td>mov [6000a0], eax</td>
</tr>
<tr>
<td>400105</td>
<td>jmp 0x40020d</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>40020d</td>
<td>mov [6000a4], 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6000a0</td>
<td>.long 0xc0debeef</td>
</tr>
<tr>
<td>6000a4</td>
<td>.long 0x0</td>
</tr>
</tbody>
</table>
Disassemble

```
.text
    mov [data_0], eax
    jmp target
    ...

.target
    mov [data_1], 1

.data
    data_0 .long 0xc0debeef
    data_1 .long 0x0
```
Patch & Assemble

Non-relocatable Assembly

<table>
<thead>
<tr>
<th>Address</th>
<th>.text</th>
<th></th>
<th></th>
<th>.data</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>400100</td>
<td>mov</td>
<td>[6000a0],</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40020d</td>
<td></td>
<td></td>
<td>CRASH!</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40020f</td>
<td>mov</td>
<td>[6000a4],</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000a0</td>
<td>&quot;cat\x00&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000a4</td>
<td>.long</td>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000a8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Fish’17]
Patch & Assemble

```
.text
mov [data_0], eax
jmp target
...

mov [data_1], 1

.data
.data
.data
.data
.data
.data
long 0xc0debeef
long 0x0
```

Relocatable Assembly

[Fish’17]
Code regions

- .text
- .rodata
- .data
- .bss

Data regions
Fall 1399

Ce 874 - Reassembly

[Fish’17]
Problem

False Positives

Hey, this is a value, not a pointer!

False Negatives

Man, this is absolutely a pointer. Why can't you tell?
Problem: Value Collisions

/* stored at 0x8060080 */
static float a = 4e-34;

A Floating-point Variable a

8060080 .db 3d
8060081 .db ec
8060082 .db 04
8060083 .db 08

Byte Representation

8060080 label_804ec3d

Interpreted as a Pointer

False Positives
Problem: Compiler Optimization

```c
int ctrs[2] = {0};

int main()
{
    int input = getchar();
    switch (input - 'A')
    {
        case 0:
            ctrs[input - 'A']++;
            break;
        ...
    }
}
```

A code snippet allows **constant folding**
Problem: Compiler Optimization

A code snippet allows constant folding

int ctrs[2] = {0};

int main()
{
    int input = getchar();
    switch (input - 'A')
    {
        case 0:
            ctrs[input - 'A']++;
            break;
        ...
    }
}

0x804a034 – ‘A’ * sizeof(int) = 0x8049f30

False Negatives

0x804a034 does not belong to any section

Compiled in Clang with –O1

[Fish’17]
Pipeline

CFG Recovery

Content Classification

Symbolization & Reassembly

0x804850b
0xa
0xdc5
63 61 74 00
0x80484a2
0x804840b
0xa0000
0x8048450b
0xa
0xdc5
63 61 74 00
0x80484a2
0x804840b
0xa0000

push offset label_34
push offset label_35
cmp eax, ecx
jne label_42
.label_42:
mov eax, 0x12fa9e5
...

[Fish’17]
Pipeline

CFG Recovery

Content Classification

Symbolization & Reassembly

push offset label_34
push offset label_35
cmp eax, ecx
jne label_42
.label_42:
mov eax, 0x12fa9e5
...

[Fish’17]
CFG Recovery

31 ed 5e
89 e1 83
e4 f0 50
54 52 68
00 25 05
08

0x80486f0:
xor ebp, ebp
pop esi
mov ecx, esp
and esp, 0xffffffff
push eax
push esp
push edx
...

Recursive Disassembly
Iterative Refinement
Content Classification

A Typical Pointer

\[ *(((\text{int}*)0x8045010) \]

A Typical Value

\[ (value \ast 42) \land 5 / 3 \]

[Fish’17]
## Content Classification

<table>
<thead>
<tr>
<th>Type Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types</td>
<td>Pointers, shorts, DWORDs, QWORDs, Floating-point values, etc.</td>
</tr>
<tr>
<td>Strings</td>
<td>Null-terminated ASCII strings, Null-terminated UTF-16 strings</td>
</tr>
<tr>
<td>Jump tables</td>
<td>A list of jump targets</td>
</tr>
<tr>
<td>Arrays of primitive types</td>
<td>An array of pointers, a sequence of integers</td>
</tr>
</tbody>
</table>

Data Types that Ramblr Recognizes
Content Classification

MOVe Scalar Double-precision floating-point value

movsd xmm0, ds:0x804d750
movsd xmm1, ds:0x804d758

Two floating-points
804d750 Floating point integer
804d758 Floating point integer

Recognizing Types during CFG Recovery
Content Classification

Recognizing Types with Slicing & VSA

chr = _getch();
switch (i)
{
    case 1:
        a += 2;
        break;
    case 2:
        b += 4;
        break;
    case 3:
        c += 6;
        break;
    default:
        a = 0; break;
}

switch (i)
{
    case 1:
        ...
    case 2:
        ...
    case 3:
        ...
    default:
        ...
}
Content Classification

Recognizing Types with Slicing & VSA

if(i > 3)
    jmp table[i * 4]

i = [0, 2] with a stride of 1

<table>
<thead>
<tr>
<th>A jump table of 3 entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>table[0]</td>
</tr>
<tr>
<td>table[1]</td>
</tr>
<tr>
<td>table[2]</td>
</tr>
</tbody>
</table>

[Fish’17]
A code snippet allows **constant folding**

 Compiled in Clang with –O1

```
int ctrs[2] = {0};

int main()
{
    int input = getchar();
    switch (input - 'A')
    {
        case 0:
            ctrs[input - 'A']++;
            break;
        ...
    }
}

; Assuming ctrs is stored at 0x804a034
; eax holds the input character
; ctrs[input - 'A']++;
    add 0x8049f30[eax * 4], 1
...

.bss
804a034: ctrs[0]
804a038: ctrs[1]

0x8049f30 does not belong to any section
```
Base Pointer Reattribution

The Slicing Result

; Assuming `ctrs` is stored at 0x804a034
; `eax` holds the input character
; `ctrs[input - 'A']++;
add 0x8049f30[eax * 4], 1
...

.bss
804a034: ctrs[0]
804a038: ctrs[1]

0x8049f30 does not belong to any section

Compiled in Clang with –O1

False Negatives

Constant un-folding

Belongs to .bss

0x8049f30

'A' * 4

0x804a034
Safety Heuristics: Data Consumer Check

Unusual Behaviors Triggering the Opt-out Rule
Symbolization & Reassembly

Symbolization

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400010</td>
<td>label_34</td>
</tr>
<tr>
<td>0x400020</td>
<td>label_35</td>
</tr>
<tr>
<td>0x400a14</td>
<td>label_42</td>
</tr>
</tbody>
</table>

Assembly Generation

```
push    offset label_34
push    offset label_35
cmp     eax, ecx
jne     label_42

.label_42:
mov     eax, 0x12fa9e5
...
```
## Data sets

<table>
<thead>
<tr>
<th></th>
<th>Coreutils 8.25.55</th>
<th>Binaries from CGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>106</td>
<td>143</td>
</tr>
<tr>
<td>Compiler</td>
<td>Clang 4.4</td>
<td>CGC 5</td>
</tr>
<tr>
<td>Optimization levels</td>
<td>O0/O1/O2/O3/Os/Ofast</td>
<td></td>
</tr>
<tr>
<td>Architectures</td>
<td>X86/AMD64</td>
<td>X86</td>
</tr>
<tr>
<td>Test cases</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Total binaries</td>
<td><strong>1272</strong></td>
<td><strong>725</strong></td>
</tr>
</tbody>
</table>
Brief Results: Success Rate

![Success Rate Chart]

- **Uroboros**
- **Ramblr**
- **Ramblr Fast**

[Fish’17]
Ramblr is the foundation of ...

- Patching Vulnerabilities
- Obfuscating Control Flows
- Optimizing Binaries
- Hardening Binaries
Another related work

Acknowledgments/References (1/2)

• [Wang’15] Reassembleable Disassembling (Slides), Shuai Wang, Pei Wang, and Dinghao Wu, Usenix Security 2015
• [Fish’17] Ramblr: Making Reassembly Great Again (Slides), Ruoyu “Fish” Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna, NDSS 2017
Acknowledgments/References (2/2)
