Hanging on a ROPe

Pablo Solé

Immunity
From a crash to a working exploit

- What do we used to do after EIP was controlled?
- Why do we need ROP?
- ROP 101 (or the infinite wheel of pain...)
- Problems of the manual approach
- An automatic answer
- Gadgets as SMT formulas
- ROP from mini-ASM
- Summary
- Conclusions
What do we used to do after EIP was controlled?

<table>
<thead>
<tr>
<th>Stack Memory</th>
<th>Stack Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Buffer</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Saved Frame</td>
<td>Garbage</td>
</tr>
<tr>
<td>Return Address</td>
<td>JMP ESP</td>
</tr>
<tr>
<td>Func Args</td>
<td>SHELLCODE</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
Why do we need ROP?
Data Execution Prevention

Stack Memory

Buffer

Garbage

JMP ESP

SHELLCODE

stack and heap are not executable anymore!
ROP 101
or the infinite wheel of pain...

Stack Swapping

MOV ECX, DWORD PTR [EAX]    XCHG EAX, ESP
LEA EDX, DWORD PTR [EBP+8]   RETN
PUSH EDX
PUSH DWORD PTR [EBP+10]
PUSH DWORD PTR [EBP+C]
PUSH EAX
CALL DWORD PTR [ECX+C]
ROP 101
or the infinite wheel of pain...

Stack Memory

<table>
<thead>
<tr>
<th>Addr-POP EAX</th>
<th>VALUE for EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr-POP ECX</td>
<td>VALUE for ECX</td>
</tr>
<tr>
<td>Addr-SUB r1,r2</td>
<td>...</td>
</tr>
</tbody>
</table>

POP EAX RETN
POP ECX RETN
SUB EAX,ECX RETN
Problems of the manual approach

- Finding the correct stack swap sequence can be hard.
- Finding the correct gadgets that we need can be hard.
- Bypassing badchars while you try to find your gadgets is difficult.
- Even for simple examples can be a very time consuming task.
Previous Automatic Approaches

• Scanning for very simple/known sequences
  – mov edx, [ecx]; ret;

• Expression trees/matching
  – WOOT '10, Dullien, Kornau, Weinmann
Problems with Previous Automatic Approaches

- Simple scanning
  - Imprecise
  - False positives and false negatives
- Expression trees
  - Possibly it might miss some semantically equal gadgets
An automatic answer

• Provided I know where my controlled buffer is in memory, what if I could find a stack-swap gadget automatically?

• What if I could create a ROP chain from some easy programming language?

Both problems can be solved using the same tool.
ROP via SMT Formula Solving
SAT & SMT

• Boolean satisfiability problem
  - \(((x \lor y) \lor ! (z \land y))\)
  - Is there a variable assignment that makes the formula TRUE
  - Solving this automatically

• SMT solvers
  - Allow higher order logics to be handled e.g. linear arithmetic, equality logic and so on
  - \((x + y = 8 \land y = 2 \land x < 4)\)
  - Many libraries and tools freely available to handle this e.g. CVC3
Our Solution

• Convert instruction sequences to SMT formulae
  – Gives a precise representation of instruction semantics
• For each ROP-shellcode instruction build another formula that gives our requirements e.g. 'EAX = [ECX] and EDX is not modified'
• To find a gadget we append our requirements and check for satisfiability/validity using a solver
x86 Instructions as SMT Formulae

• For each instruction in a gadget we need to convert it to an SMT formula
• add eax, ebx ->
  - regs['eax'] = solver.addExpr(regs['EAX'],
    regs['EBX'])
  - flags['_CF'] = ...

Gadgets as SMT Formulae

- At analysis time we iterate over the instructions and build the conjunction of each sub-formula

\[
\begin{align*}
\texttt{add eax, ebx} \\
\texttt{sub eax, [ecx]} \\
\quad \text{solver.subExpr(} \\
\quad \quad \text{solver.addExpr(regs['EAX'], regs['EBX'])),} \\
\quad \quad \text{mem[regs['ECX']])}
\end{align*}
\]

(Accounting for flags as well)
Finding Gadgets Using a Solver

• What defines a useful gadget?
  – Its semantics meet some criteria e.g. 'I want the value EAX+4 to be in ESP. Please don't mangle EDX while you're at it'
  – These requirements are easily expressed as SMT formula
    
    \[
    \begin{align*}
    \text{ESP}_\text{after} &= \text{EAX}_\text{before}+4 \wedge \\
    \text{EDX}_\text{after} &= \text{EDX}_\text{before}
    \end{align*}
    \]
  – Using a solver we can then query the status of \text{GADGET\_FORMULA} \wedge \text{REQUIREMENTS}
Satisfiability & Validity

• A solver can tell us if a formula is satisfiable or valid
• Satisfiability – There exists at least one variable assignment that makes the formula TRUE
• Validity – There exists no variable assignment that makes the formula false
Generic & Context Specific Gadgets

• A formula that is valid implies that regardless of memory/register context it meets our requirements
  – The gadget will always do what we want

• A formula that is satisfiable but not valid will meet our requirements under certain conditions
  – It will do what we want given certain preconditions on registers and memory
Workflow

- Find candidate gadgets
- From each gadget build an SMT formula $G$
- For each ROP primitive build a SMT formula $R$
  - More on this later (mini-asm)
- For every $r$ in $R$ and $g$ in $G$ build $(r \land g)$ and check for satisfiability or validity (depending on your requirements)
Implementation
Find Gadget Candidates

- Search RETN opcodes (0xC2 or 0xC3) in the entire DLL memory
- Disassemble backward until it finds an unsupported/invalid opcode
- Generate all possible disassemblies (move a byte and magic can occur)
- Finally, it returns lists of opcodes for each RETN-ended sequence
# Candidates Example

## Binary Data

<table>
<thead>
<tr>
<th>Segment</th>
<th>Address</th>
<th>Bytes</th>
<th>Value</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7C91990D</td>
<td>66 83 26 00</td>
<td>66 83 66</td>
<td>ff&amp;.fff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C919915</td>
<td>02 00 83 66</td>
<td>04 00 5E 5D</td>
<td>.ff.^]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991D</td>
<td>C2 04 00</td>
<td></td>
<td>Â.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Possible Disassemblies

<table>
<thead>
<tr>
<th>Segment</th>
<th>Address</th>
<th>Bytes</th>
<th>Value</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7C91991A</td>
<td>00 5E 5D</td>
<td></td>
<td>ADD BYTE PTR DS:[ESI+5D],BL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991D</td>
<td>C2 0400</td>
<td></td>
<td>RETN 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C919919</td>
<td>04 00</td>
<td></td>
<td>ADD AL,0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991B</td>
<td>5E</td>
<td></td>
<td>POP ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991C</td>
<td>5D</td>
<td></td>
<td>POP EBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991D</td>
<td>C2 0400</td>
<td></td>
<td>RETN 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91990E</td>
<td>66 83 26 00</td>
<td></td>
<td>AND WORD PTR DS:[ESI],0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C919912</td>
<td>66 83 66 02 00</td>
<td></td>
<td>AND WORD PTR DS:[ESI+2],0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C919917</td>
<td>83 66 04 00</td>
<td></td>
<td>AND DDWORD PTR DS:[ESI+4],0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991B</td>
<td>5E</td>
<td></td>
<td>POP ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991C</td>
<td>5D</td>
<td></td>
<td>POP EBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C91991D</td>
<td>C2 0400</td>
<td></td>
<td>RETN 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequence Analyzer

- Emulate each instruction
- Generate a resulting CPU/Memory context
- Support interactions between CPU and Memory
- Use SMT Expressions for the emulation
- Support abstract memory addressing
  - MOV EAX, DWORD PTR DS:[EDX]
    (given we don't know the final value of EDX)
- It's easy to add new architectures (x64, arm, etc).
Sequence Analyzer

0100739D > 33C0 XOR EAX,EAX
0100739F 03C2 ADD EAX,EDX
010073A1 3BC2 CMP EAX,EDX
010073A3 74 05 JE SHORT 010073AA
010073A9 33C0 XOR EAX,EAX
010073AA 03C3 ADD EAX,EBX
Gadget Properties

• We calculate a set of properties like: what registers were read, written or dereferenced.

• This properties are used as a first criteria for gadget searching.

• The smarter we are discovering this properties, the faster we're going to find a useful gadget.
Gadget Complexity Index

- How complex is this gadget?
  - how many registers does it modify?
  - how many memory operations does it have?
  - how much has the stack pointer moved?

MOV EDI,EDI
RETN

MOV EDI,EAX
POP EAX
POP EBX
RETN 4

MOV EAX,[EBX]
POP ECX
RETN 0C

XOR EDI,[EBX+ECX*4]
MOV [EDI], EAX
XOR EAX,EAX
POP ECX
RETN 30

COMPLEXITY
Use cases and more details...
Stack swapping

• What does that means?

1) ESP = Controlled Memory Address
2) EIP = Controlled Memory Content

1) XCHG EAX,ESP
   MOV EAX, [EAX]
   MOV [ESP], EAX
   RETN

2) MOV ESI, [EAX]
   CALL ESI

1 & 2) XCHG EAX,ESP
       RETN
Stack swapping

1) ESP = Controlled Memory Address
2) EIP = Controlled Memory Content

• On a SMT formula:
  (EAX = address controlled mem)
  – 1) eqExpr(ESP, EAX+4)
  – 2) eqExpr(EIP, mem(EAX))
  – 1 & 2) boolAndExpr(1, 2)
Stack swapping
ROP from mini-ASM

• We need a kind of ROP “compiler”
• Some of its responsibilities:
  – Alloc/Release registers
  – Preserve stack memory from accidental overwrites.
  – Satisfy gadget pre-conditions
  – Find the best way of performing a mini-ASM instruction.
  – Bypass badchars
  – Create the final ROP chain
ROP from mini-ASM

We can use many tricks to implement an instruction!

Let's say we want to **MOV EAX, 0x1234**

- POP EAX
- RETN
- MOV EAX,1234
- RETN
- EBX points to some place in our ROP chain
- MOV EAX,[EBX]
- RETN
- POP EAX
- RETN
- POP ECX
- RETN
- SUB EAX,ECX
- RETN
ROP from mini-ASM

- Lets say we have 4 tricks for storing a value in register.

- In DEPLIB we associate handlers for each instruction, where we implement these tricks.

- Also, each trick has a preference, so we use the shorter cases first.
ROP from mini-ASM

- From a SMT formula perspective, we just append all our gadget requirements and our guard conditions (regs/mem/flags that must be guarded)
- Ask the Solver if there's a gadget that satisfy our query.
ROP from mini-ASM

A mini-ASM example:

va_addr=solveImport("kernel32!VirtualAlloc")
args=(0,0x1000,0x3000,0x40)
allocated_buf=call(va_addr, args, callconv="stdcall")
jmp_addr=VAR()
mov(jmp_addr, allocated_buf)

shellcode_ptr=endofROP()
shell_dword=VAR()
shell_dword.bind("mem", shellcode_ptr)

label("decrypt_loop")
xor(shell_dword, 0xdeadbeef)
mov(allocated_buf, shell_dword)
add(allocated_buf, 4)
add(shellcode_ptr, 4)
ifne(shell_dword, 0xcafe, "decrypt_loop")
jmp(jmp_addr)
Summary

- We emulate the x86 instruction set using a SMT Solver (no FPU/SSE/etc.)

- Then we store a SMT representation of all registers, flags and memory accesses.

- This means we capture the semantics of a sequence of instructions.

- Our solver of choice was CVC3 Solver.
Summary

This allows us to answer some non-obvious questions. ex:

• Is there a gadget that sets
  \[ \text{ESP} = \text{EAX} + 4 \quad \text{AND} \quad \text{EIP} = [\text{EAX}] \]?

STACK SWAPPING
Summary

• Is there a gadget that sets
  EAX=value, without touching ESI,EDI?

RETURN ORIENTED PROGRAMMING
Summary

But also solve things like:

- **Is there** a value for EAX that takes a given branch? (and **what** is that value):

  IMUL EAX, ECX, 4
  SUB EAX, [EBP+10]
  CMP EAX, 100
  JL allowed

**SYMBOLIC EXECUTION**
Conclusions

• DEPLIB 2.0 is going to be part of the release of **Immunity Debugger 2.0 on December 2010**.
• Lots of different tools can be made from the work presented here.
• **ROP-only** shellcode on x86 is possible using DEPLIB.
• Concepts of ROP can be extended to other code reuse techniques:
  – Chain gadgets using jumps
  – Chain gadgets using calls
Thank you for your time

Contact me at:
pablo.sole@immunityinc.com