Method for Analysis of Code-reuse Attacks
Reverse Engineering of ROP Exploits

Alexey Vishnyakov
Alexey Nurmukhametov
Shamil Kurmangaleev
Sergey Gaisaryan
23 November 2018

ISP RAS
Vulnerabilities by Year

Number (tens of thousands) of new vulnerabilities (CVE) by year

Source: cvedetails.com/browse-by-date.php
Motivation

- Deliberate exploitation of vulnerabilities can lead to information disclosure, financial losses, or even greater damage.
- Big companies perform computer security incidents analysis.
- Return-oriented programming (ROP) is an exploitation technique that can be used in presence of modern operating systems protections.
- The main contribution of our work is to simplify ROP exploits reverse engineering.
• **Buffer Overflow Vulnerability** exists when a program attempts to put more data in a buffer than it can hold

• Buffer overflow causes a **return address overwrite**
Stack Smashing:
- Place payload on the stack
- Overwrite return address with a pointer to the payload
- Execute arbitrary code

Executable Space Protection:
- **Executable space protection** (DEP) marks memory regions as non-executable
- In particular, the execution of malicious code placed on the stack is forbidden
Return-to-libc Attack

Return-to-libc attack bypasses DEP:

- Overwrite return address with a library function address, for instance, `system`
- Prepare function arguments on the stack
Address Space Layout Randomization

- **Address space layout randomization (ASLR)** is an operating system protection that randomly arranges the address space positions of key data areas of a process (base of the executable, stack, heap, dynamic libraries)
- Library function address is unknown before the program load
- Modern ASLR implementations leave some program address space areas **non-randomized**:
  - In Linux the base of the executable is often left constant
  - Some Windows dynamic libraries are loaded at constant offsets
Return-oriented Programming

- **Return-oriented Programming (ROP)** is a code-reuse attack that allows an attacker to bypass DEP in presence of non-randomized memory areas.
- Attacker uses gadgets – code blocks from non-randomized memory address space.
- Each gadget performs some computation (for instance, adds two registers) and transfers control to the next gadget.
- Gadgets are chained together and executed consequently.
- Thus, a gadget chain executes a malicious payload.
ROP gadgets

- **Gadget** is an instruction sequence – in non-randomized executable memory area – that ends with a control transfer instruction (usually with `ret`)

- Because x86 architecture doesn’t require instruction aligning, an instruction sequence can contain a gadget that is not present in original program code:

  \[
  \text{f7c7070000000f9545c3 → test edi, 0x7 ; setnz BYTE PTR [ebp-0x3d]} \\
  \text{c7070000000f9545c3 → mov DWORD PTR [edi], 0xf0000000 ; xchg ebp, eax ; inc ebp ; ret}
  \]

- Gadget addresses are placed on the stack starting from the return address so that the first gadget transfers control to the second one, the second one – to the third one, and so on

*Jonathan Salwan. An introduction to the Return Oriented Programming and ROP chain generation*
### ROP Chain Example

**Write** `memValue` **to** `memAddr`

<table>
<thead>
<tr>
<th>...</th>
<th>mov [edx], eax ; ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th gadget address</td>
<td></td>
</tr>
<tr>
<td>3rd gadget address</td>
<td></td>
</tr>
<tr>
<td><code>memAddr</code></td>
<td>pop edx ; ret</td>
</tr>
<tr>
<td>2nd gadget address</td>
<td></td>
</tr>
<tr>
<td><code>memValue</code></td>
<td>Previous return address location</td>
</tr>
<tr>
<td>1st gadget address</td>
<td>pop eax ; ret</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
ROP Chain is a Program

- ROP chain is a program for a virtual machine defined by an executable
- Stack pointer acts as a program counter
- Instruction **opcodes** (gadget addresses) and **operands** are placed on the stack

Virtual machine instructions:
- `mov [edx], eax`
- `mov edx, memAddr`
- `mov eax, memValue`

Real instructions:
- `mov [edx], eax ; ret`
- `pop edx ; ret`
- `pop eax ; ret`
Problem Definition

Given a binary ROP chain, we should:

- Restore a gadget chain
- Determine semantics of each gadget
- Restore function calls with arguments
- Detect system calls
In order to split ROP chain into gadgets, we define a *gadget frame* similar to x86 stack frame.

- **Frame size**
  \[ \text{FrameSize} = 16 \]

- **Next gadget address**
  \[ \text{NextAddr} = [\text{ESP} + 4] \]
Gadget Semantic Definition

- **Gadget type** is defined semantically by a postcondition – a boolean predicate that must always be true after executing the gadget:
  - MoveRegG: OutReg ← InReg
  - LoadConstG: OutReg ← [SP + Offset]

- Set of gadget types is an instruction set architecture (ISA)

- Gadget function is described with a set of parameterized types that satisfy the gadget

- Gadget classification determines a set of possible types and parameters

```
PUSH EAX
POP EBX        MoveRegG: EBX ← EAX
POP ECX        LoadConstG: ECX ← [ESP + 0]
RET
```

Gadget Classification

- We perform classification after analysing effects of gadget execution on different inputs
- Gadget instructions are translated into the intermediate representation*
- Then the interpretation of intermediate representation starts
  - All memory and register accesses are tracked
  - Initial values of registers and memory areas are generated randomly
  - As a result of interpretation, the initial and final values of registers and memory will be obtained
- We perform several more interpretations with different inputs and gather a list of types and parameters with true postconditions for all executions

ROP Chain Semantics Analysis

- Binary ROP chain is loaded onto the shadow stack
- Gadgets are classified one by one according to frame info
- Shadow memory is used to restore values of registers and memory before functions and system calls
  - Initially, a shadow memory is empty
  - We perform several interpretations of gadget with a shadow memory as an initial state
  - Final values of registers and memory – unchanged from execution to execution – are added to shadow memory
• Names of **indirect** function calls are gathered from import tables
  
  ```asm
  JMP [EAX]
  ```

• Linux system calls and functions prototypes can be found in man-pages

• System call number and arguments are gathered from the shadow memory
Example: MongoDB Linux x86 (CVE-2013-1892)

Binary representation of the ROP chain:

```
00000000  68  f7  16  08  07  6d  66  08  00  70  33  31  00  20  00  00  |h....mf..p31.  ..|
00000010  07  00  00  00  31  00  00  00  ff  ff  ff  ff  00  00  00  00  |.....1...........|
00000020  00  00  00  00  c8  e4  16  08  00  70  33  31  00  70  33  31  |.........p31.p31|
00000030  00  00  0b  0c  00  20  00  00  |...... ..|
00000038
```
Example: MongoDB Linux x86 (CVE-2013-1892)

0x0816f768 : Asm : JMP DWORD PTR [08A1AF84h]
0x0816f768 : Call [0x8a1af84]
0x0816f768 : mmap(0x31337000, 0x2000, 0x7, 0x31, 0xffffffff, 0x0) from libc.so.6

0x08666d07 : Asm : ADD ESP, 00000014h ; POP EBX ; POP EBP ; RET
0x08666d07 : LoadConstG : EBX <- [ESP+20], EBP <- [ESP+24] :
               NextAddr=[ESP+28], FrameSize=32
0x08666d07 : ShiftStackG : ESP +<- 28
0x08666d07 : Values : EBX <- 0x0 ("\x00\x00\x00\x00"),
               EBP <- 0x0 ("\x00\x00\x00\x00")

0x0816e4c8 : Asm : JMP DWORD PTR [08A1AADCh]
0x0816e4c8 : Call [0x8a1aadc]
0x0816e4c8 : memcpy(0x31337000, 0xc0b0000, 0x2000) from libc.so.6
0x31337000 : Call 0x31337000
0x31337000 : Values : [ESP+4] <- 0xc0b0000, [ESP+8] <- 0x2000
## Results

<table>
<thead>
<tr>
<th>Application</th>
<th>CVE Number</th>
<th>Platform</th>
<th>Gadgets from</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>CVE-2013-1892</td>
<td>Linux x86</td>
<td>mongod</td>
</tr>
<tr>
<td>Nagios3</td>
<td>CVE-2012-6096</td>
<td>Linux x86</td>
<td>history.cgi</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>CVE-2010-4221</td>
<td>Linux x86</td>
<td>proftpd</td>
</tr>
<tr>
<td>Nginx</td>
<td>CVE-2013-2028</td>
<td>Linux x64</td>
<td>nginx</td>
</tr>
<tr>
<td>AbsoluteFTP</td>
<td>CVE-2011-5164</td>
<td>Windows x86</td>
<td>MFC42.dll</td>
</tr>
<tr>
<td>ComSndFTP</td>
<td>N/A 2012-06-08</td>
<td>Windows x86</td>
<td>msvcrtd.dll</td>
</tr>
</tbody>
</table>
Extra
Gadget Verification

- Gadget classification provides a set of postconditions describing possible gadget semantics.
- Gadget verification formally proves these postconditions for each input.
- Gadget verification implementation is based on Triton dynamic symbolic execution engine.
  - Initially, all registers are assigned to free symbolic variables.
  - Symbolic memory is implemented via select and store operations over SMT array.
  - Symbolic execution of gadget instructions generates SMT formulas over constants and variables, it also updates the symbolic state of registers and memory.
  - Postcondition validity is checked via unsatisfiability of its negation.

Triton: github.com/JonathanSalwan/Triton
# Gadget Verification Example

**ArithmeticLoadG**: \( rbx \leftarrow rbx + [rax] \)

<table>
<thead>
<tr>
<th>Step</th>
<th>Symbolic state</th>
<th>Instruction</th>
<th>Set of symbolic expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>( M, rax = \phi_1, rbx = \phi_2, rcx = \phi_3, rsp = \phi_4, rip = \phi_5 )</td>
<td>—</td>
<td>( S_0 = \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( rcx = \phi_6 )</td>
<td>mov rcx, [rax]</td>
<td>( S_1 = S_0 \cup { \phi_6 = M[\phi_1] } )</td>
</tr>
<tr>
<td>2</td>
<td>( rbx = \phi_7 )</td>
<td>add rbx, rcx</td>
<td>( S_2 = S_1 \cup { \phi_7 = \phi_2 + \phi_6 } )</td>
</tr>
<tr>
<td>final</td>
<td>( rip = \phi_8, rsp = \phi_9 )</td>
<td>ret</td>
<td>( S_3 = S_2 \cup { \phi_8 = M[\phi_4], \phi_9 = \phi_4 + 8 } )</td>
</tr>
<tr>
<td><strong>Semantic definition</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>verify</td>
<td>((\text{final}(rbx) = \text{initial}(rbx) + \text{initial}(M[rax])) \land )</td>
<td>(~((\phi_7 = \phi_2 + M[\phi_1]) \land (\phi_8 = M[\phi_4]) \land (\phi_9 = \phi_4 + 8)) \text{ is UNSAT} )</td>
<td></td>
</tr>
</tbody>
</table>


Semantic verification