Defeating DEP, the Immunity Debugger way

Pablo Solé

Immunity
Old School Stack Overflow

Stack Memory

<table>
<thead>
<tr>
<th>Buffer</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Saved Frame</td>
<td></td>
</tr>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Func Args</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

Stack Memory

<table>
<thead>
<tr>
<th>Buffer</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Garbage</td>
<td></td>
</tr>
<tr>
<td>JMP ESP</td>
<td></td>
</tr>
<tr>
<td>SHELLCODE</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
Data Execution Prevention

Stack Memory

Buffer
.
.
.
Garbage

JMP ESP

SHELLCODE
.
.
.
.

stack and heap are not executable anymore!
Let's say we want to call a function that takes the arguments from registers EAX and ECX.
Survey of the Landscape

Data Execution Prevention

vs

Immunity Debugger

What I Have

CPU/Memory Context Control

What I Want

EAX=alloc(0x1000, RWX)
memcpy(EAX, shellcode)
jmp(EAX)

What I Need

Stack String (that bypass filters)
Some Previous Efforts

• EEREAP by eEye
  – made a cpu/memory emulator using process snapshots
• Return-Oriented Programming by Hovav Shacham
  – presented a Turing-complete language using pre-selected opcode sequences (gadgets)
• manual efforts
  – you can always search your own ret-to-libc pieces manually
An Aside

I need a pop/pop/ret FTW!

EIP = [ESP + 8]

POP R32
POP R32
RETN

ADD ESP, 8
RETN 30

XCHG EAX, ESP
LEA EAX, [EAX+8]
MOV EDX, [EAX]
XCHG EAX, ESP
RETN

DEPLib.seteip(['ESP','+',8])

Hacker's needs
(Lazy Level)

Mind (Theoretical Level)

Reality (Binary level)

Magic! (DEPLib level)
Real World Problems

- Win32, not Linux
- Cannot be limited to libc (or any particular library)
- Optimize for filter bypassing and size
- Simplistic language
- It should find the necessary sequences automatically
Many pieces combining = VOLTRON

- SearchDEP
- DEPLib Parser/Generator
- DEPLib DB
- Sequence Analyzer
- Pieces Analyzer
SearchDEP

• 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 can occur)
• Finally, it returns lists of opcodes for each RETN-ended sequence
SearchDEP Example

Binary Data

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex</th>
<th>Hex</th>
<th>Hex</th>
<th>Hex</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>66 83 26 00</td>
<td>66 83 66</td>
<td>ff&amp;.fff</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0400</td>
<td>02 00 83 66 04 00</td>
<td>5E 5D</td>
<td>.ff.^]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td>C2 04 00</td>
<td></td>
<td>Å.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible Disassemblies

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>005E 5D</td>
<td>ADD BYTE PTR DS:[ESI+5D],BL</td>
</tr>
<tr>
<td>0x0400</td>
<td>C2 0400</td>
<td>RETN 4</td>
</tr>
<tr>
<td>0x0800</td>
<td>04 00</td>
<td>ADD AL,0</td>
</tr>
<tr>
<td>0x0C00</td>
<td>5E</td>
<td>POP ESI</td>
</tr>
<tr>
<td>0x0C10</td>
<td>5D</td>
<td>POP EBP</td>
</tr>
<tr>
<td>0x0C18</td>
<td>C2 0400</td>
<td>RETN 4</td>
</tr>
<tr>
<td>0x0C20</td>
<td>668326 00</td>
<td>AND WORD PTR DS:[ESI],0</td>
</tr>
<tr>
<td>0x0C28</td>
<td>668366 02 00</td>
<td>AND WORD PTR DS:[ESI+2],0</td>
</tr>
<tr>
<td>0x0C30</td>
<td>8366 04 00</td>
<td>AND DWORD PTR DS:[ESI+4],0</td>
</tr>
<tr>
<td>0x0C38</td>
<td>5E</td>
<td>POP ESI</td>
</tr>
<tr>
<td>0x0C40</td>
<td>5D</td>
<td>POP EBP</td>
</tr>
<tr>
<td>0x0C48</td>
<td>C2 0400</td>
<td>RETN 4</td>
</tr>
</tbody>
</table>
Sequence Analyzer

- Emulate each instruction
- Generate a resulting CPU/Memory context
- Support interactions between CPU and Memory
- Solve modulo $2^8/16/32$ arithmetic
  - $a \xor a = 0$ / $a \text{ and } 0 = 0$
  - $a \xor \neg a = \text{all-ones}$
  - $(a >> 16) \text{ and } \neg(a >> 16) = 0$
- Support abstract memory addressing
  - MOV EAX, DWORD PTR DS:[EDX]
    (supposing we don't know EDX value)
CPU/Memory Context
Pieces Analyzer

- Summarize the consequences of executing a sequence over our running context
- Make hashes of each CPU/Memory change
- Calculate a complexity value of each piece
- We don't need to deal with instructions anymore, just CPU/Memory state
Consequences

- Is a change is the CPU/Memory context
- Each piece generate a set of consequences
- We can search in our DB for these consequences using the hashes

\[
\text{EBX: } [[\text{'='}, \text{ ['con', 0L]}, 32L, 0L]]: \text{C59756C0}
\]

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>CRC32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/12/08
Piece Complexity Index

• How complex is this piece?
  – how many consequences does it have?
  – how many memory operations does it have?
  – how much has the stack pointer moved?

| MOV EDI,EDI | MOV EDI,EAX | MOV EAX,[EBX] | XOR EDI,[EBX+ECX*4] |
| MOV EAX, [EBX] | POP EAX | POP ECX | MOV [EDI], EAX |
| POP EBX | RETN 0C | XOR EAX,EAX | POP ECX |
| RETN 4 | RETN 4 | RETN 0C | RETN 30 |

COMPLEXITY
ID Database

- Store all module's pieces along with necessary information to replay the sequence
- Using the consequence hash we can find suitable pieces quick and easy
- And we always get the simpler piece that does the job thanks to the complexity index
The DB

```
SELECT * FROM pieces WHERE piece_id IN
(SELECT piece_id FROM consequences WHERE consequence_hash = "32D7A775")
ORDER BY piece_complexity
LIMIT 5
```

<table>
<thead>
<tr>
<th>piece_id</th>
<th>piece_complexity</th>
<th>module_id</th>
<th>size</th>
<th>module_offset</th>
<th>piece_dump</th>
<th>piece_properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5921</td>
<td>BLOB (Size: 1077)</td>
<td>['LoadStack', 'LoadReg']</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4626</td>
<td>BLOB (Size: 1077)</td>
<td>['LoadStack', 'LoadReg']</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4624</td>
<td>BLOB (Size: 1077)</td>
<td>['LoadStack', 'LoadReg']</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4622</td>
<td>BLOB (Size: 1077)</td>
<td>['LoadStack', 'LoadReg']</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4620</td>
<td>BLOB (Size: 1077)</td>
<td>['LoadStack', 'LoadReg']</td>
</tr>
</tbody>
</table>
Data Data everywhere

- NTDLL generates \( \sim 10,000 \) pieces
- Many pieces have the same consequences (but we need them to bypass filters)
- Processing an average sized library takes 4 hrs
- SQL allows us to make arbitrary complex queries
  - Find pieces with the same address over a set of modules (universal addresses)
  - Use only application specific addresses (independent from the OS)
DEPLib Parser

- Track register use to avoid undesirable overwriting of already settled values (du chains)
- It supports variables
- It does register reusing over non-overlapped variables
- Calculate all possible combinations of variable to register mapping
About the Language

• Almost assembler:
  – mov, sub, add
  – xor, and, or
  – shl, shr, rol, ror
  – call (using fixed or dynamic stack arguments)

• And some useful additions:
  – jump to your shellcode
  – find your stack stream
    • parser.findbuffer(['EAX','+','8'])
      means: EIP=[EAX+8], ESP=EAX+12
  – find the stack stream end
DEPLib Parser Example

```python
def test(self):
    imm = immLib.Debugger()
    parser = DEPLibParser(imm)

    parser.mov("EDX", 0x00040000)  # f10ptions (HEAP_CREATE_ENABLE_EXECUTE)
    parser.mov("ECX", 0x00001000)  # dwInitialSize
    parser.mov("EAX", 0)          # dwMaximumSize
    parser.call_args("KERNEL32.HeapCreate", 3)
```
Transformation...

```plaintext
reg:EAX, defines:[2L, 3L]
reg:EDX, defines:[0L]
reg:ECX, defines:[1L]
reg:EAX, uses:[3L]
reg:EDX, uses:[3L]
reg:ECX, uses:[3L]
reg:EAX, frees:[3L]
reg:EDX, frees:[3L]
reg:ECX, frees:[3L]
reg:EAX, needed:[]
reg:EDX, needed:[1L, 2L]
reg:ECX, needed:[2L]

cmdpos: 0, cmd:['MOV', [['reg', 'EDX'], ['const', 262144L]]]
regs:[], vars:[], defregs:['EDX'], defvars:[]

cmdpos: 1, cmd:['MOV', [['reg', 'ECX'], ['const', 4096L]]]
regs:['EDX'], vars:[], defregs:['ECX'], defvars:[]

cmdpos: 2, cmd:['MOV', [['reg', 'EAX'], ['const', 0L]]]
regs:['EDX', 'ECX'], vars:[], defregs:['EAX'], defvars:[]

cmdpos: 3, cmd:['CALL_ARGS', [['const', 2088840262L], 3L, ['EAX', 'ECX', 'EDX'], False]]
regs:[], vars:[], defregs:['EAX'], defvars:[]
```
DEPLib Generator

• Searches in the database for pieces that generate our desired consequences (we don't use hand-selected addresses, all is done dynamically)

• Checks that each piece satisfies a set of preconditions:
  – undesired memory writing/reading
  – undesired register overwriting
  – piece effective address bypass chars filtering

• Creates a stack sequence that we need and fills the blanks with good chars
DEPLib Parameters

• To start generating our stack stream, we need the following information from the user:
  – a DEPLib Parser instance
  – a list of allowed modules to get the pieces from
    • OS specific?
    • Application specific?
    • Universal addresses?
  – a list of memory addresses where we can read or write (optional)
  – a list of bad chars (to bypass chars filtering)
Some tricks...

- We need to support loading of arbitrary values to registers (even if they have bad chars)

```python
def mov(reg, value):
    if value has badchars:
        PopPopSubTrick(reg, value)
    else:
        PopTrick(reg, value)

...  

def PopPopSubTrick(reg, value):
    (val1, val2) = findSubValues(value)
    mov(reg, val1)
    reg2 = findFreeReg()
    mov(reg2, val2)
    sub(reg, reg2)
```

DEPLib Language Level

DEPLib Logic Level
Some tricks...

MOV EAX, 0xFFFFFFFF
MOV EDX, 0xFFF
SUB EAX, EDX

Assembler Level

Stack Memory

<table>
<thead>
<tr>
<th>77F21564</th>
<th>FFFFFFFF</th>
<th>77F33A40</th>
<th>FFFBFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>77F31293</td>
<td></td>
<td>77F31293</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Return Programming Level

<table>
<thead>
<tr>
<th>77F21564</th>
<th>58 POP EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>77F21565</td>
<td>C3 RETN</td>
</tr>
<tr>
<td>77F33A40</td>
<td>5A POP EDX</td>
</tr>
<tr>
<td>77F33A41</td>
<td>C3 RETN</td>
</tr>
<tr>
<td>77F31293</td>
<td>2BC2 SUB EAX, EDX</td>
</tr>
<tr>
<td>77F31295</td>
<td>C3 RETN</td>
</tr>
</tbody>
</table>
Metrics

- We’ve obtained a stack stream that successfully and reliably do:
  - **HeapCreate** with the Executable Page option
  - **Allocate** a chunk in this new heap
  - **Memcpy** our shellcode
  - **Jump** to the allocated chunk

  In just **280** bytes, bypassing NULL chars filters
  If you don’t have badchars it’s **half** that size

- The smallest stream we have created (always bypassing NULL char filtering) is **236** bytes long
DEMO
Future Work

• Support the entire x86 instruction set

• Interpret flags and do conditional analysis

• Support conditional execution and looping on DEPLib to create a Turing-Complete implementation
  – Not just for the FUN, but to execute shellcode selectively
The Conclusion

Automatically defeating DEP is not just an idea is a FACT
Thank you for your time

Contact me at:
pablo.sole@immunityinc.com