Just-In-Time Code Reuse

By: Erik Hyllienmark
1.0 Background

- Information flows are growing
- Information are becoming an asset
- Many architectures and systems are today vulnerable
1.1 History

Attacks

- Stack smashing
- Code reuse attacks

Countermeasures

- Canary
- No execute bit
- Address space layout randomization
2.0 Attack

- Goal
- Gadgets
- Classic ROP attack
- Framework
- Summary
- Implementation
- Evaluation
2.2 Goal

- Hijack control flow
- Execute arbitrary code
- Exit without detection
2.3 Gadgets

- Sequence of instructions residing in memory
- Ending with a return instruction
- Can be chained together
- Performs a well-defined task such as a load-, add- or sub operation
2.4 Classic ROP Attack

- Write a payload into the application memory space
- Exploit vulnerability to hijack execution flow
- Redirect execution flow
- Direct stack pointer to beginning of the payload
2.5 Framework

- Exploits memory disclosure multiple times
- Bypasses many mitigations techniques
- Compatible with all OS revisions
- Remote exploits or local binary exploitation

Conjecture

“If fine-grained exploit mitigations became common-place, then attackers would simply modularize, automate and incorporate a similar approach into existing exploitation toolkits”
2.5.1 Assumptions

- Non-Executable memory
  - Applied to stack and heap. Applied to all executables and native system libraries.

- JIT spraying mitigations
  - Randomization JIT pages, constant variable modifications and random NOP insertions.

- Base address randomization
  - ASLR

- Fine-Grained ASLR
  - Permutates the order of functions and basic blocks
  - Swaps registers and replaces instructions
  - Randomizes the location of each instructions
  - Performs randomization on each execution of the application
2.5.2 Precondition

- Memory disclosure
- Initial code pointer
2.5.3 Workflow

1. Provide initial code pointer
2. Mapping code page memory
3. Find API function pointers
4. Find gadgets
5. JIT Compile
6. Exploit-Specific
Step 2: Mapping code page memory

- Uses initial code pointer
- Direct and indirect jmp instructions
- Avoids program crash
- Filters invalid instruction

Algorithm 1 HarvestCodePages: given an initial code page, recursively disassemble pages and discover direct and indirect pointers to other mapped code pages.

```
Input: P {initial code page pointer}, C {visited set}
Output: C {set of valid code pages}
if ∃(P ∈ C) {already visited} then
    return
end if
C(P) ← true {Mark page as visited}
\( \bar{P} = \text{DisclosePage}(P) \) {Uses DiscloseByte() internally to fetch page data}
for all ins ∈ Disassemble(\( \bar{P} \)) do
    if isDirectControlFlow(ins) then
        {e.g. JMP +0x5555}
        ptr ← ins.offset + ins.effective_address
        HarvestCodePages(ptr)
    end if
    if isIndirectControlFlow(ins) then
        {e.g. CALL -0xFeef}
        iat_ptr ← ins.offset + ins.effective_address
        ptr ← DisclosePointer(iat_ptr) {Internally uses DiscloseByte() to fetch pointer data}
        HarvestCodePages(ptr)
    end if
end for
```
Step 3: Find API function pointers

- Uses initial code pointer
- Interact with the kernel
- LoadLibrary
- GetProcAddress
Step 4: Find gadgets

- Uses initial code pointer
- Gadgets for payload
- Using code harvested in Step 2
- Combines gadgets for high-order computations
- Gadgets with side effects dismissed

---

**Algorithm 2** VerifyGadget: Automatically match a sequence of instructions to a gadget’s semantic definition.

```
Input: S {sequence of consecutive instructions}, D {gadget semantic definitions}
Output: G {gadget type, or null}
head ← S(0) {first instruction in sequence}
if G ← LookupSemantics(head) ∉ D {implemented as a single table-lookup} then
    return null
end if
for i ∈ 1...|S| {ensure semantics are not violated by subsequent instructions} do
    ins ← S(i)
    if HasSideEffects(ins) || RegsKilled(ins) ∈ RegsOut(head) then
        return null
    end if
end for
return G {valid, useful gadget}
```
Step 5: JIT compilation

- Using API functions pointers and gadgets found
- Done during program execution
- Can adapt to dynamic changes
- Can be optimized
- Necessary when fine grained exploit mitigations are used
- Compilation supports download, execute and others
Step 6: Exploit specific

- Payload is serialized to a structure accessible from the script
- Control is returned to exploit writer’s code
2.5.4 Summary

- Executable script
- Supports multiple platforms
- Speed is important
- Approximately 3000 LOC
2.6 Implementation

- Scenario: victim browses a HTML page
- Loads calc application
- Exploits memory disclosure multiple times

```javascript
// ... snip ...
// The string object is overwritten, and initial code
// pointer harvested prior to this snippet of code

// Step 1, implement DiscloseByte interface
framework.prototype.DiscloseByte = function(address) {
    var value = this.string.charCodeAtAt((address - this.absoluteAddress - 8)/2);
    if (address & 1) return value >> 8; // get upper
    return value & 0xFF; // value is 2 bytes, get lower
};

// Define target program ('8' is shorthand
// for 'last value returned')
var program =
    "LoadLibraryW(L\kernel32\);" +
    "GetProcAddress(@, 'WinExec');" +
    "@('calc', 1);" +
    "LoadLibraryW(L\kernel32\);" +
    "GetProcAddress(@, 'ExitProcess');" +
    "@(1);";

// Steps 2-4, harvest pages, gadgets, functions
framework.HarvestCodePages(this.initialCodePtr_);

// Step 5, 6 - jit-compile and build exploit buffer
var exploitBuffer =
    repeat(0x3E, unescape("%u9191%u9191")) + // Id
    repeat(0x19, framework.Nop8()) + // Sled
    unescape(framework.Compile(program)) + // Payload
    repeat(0x12, unescape("%u5454%u5454")) + // Pad
    repeat(0x32, framework.StackPivotG()); // Redirect

// overwrite with the exploit buffer
// ... snip ...
```

Figure 5. A JavaScript code sample from our proof of concept exploit illustrating each of the steps from our workflow.
2.7 Evaluation

- Code page harvesting
  - 300 pages harvested from IE
  - Pages harvested depends on initial code pointer
- Gadget coverage
  - Can generate payload from 78% of the initial code pages
- Runtime performance
  - See Figure

Figure 8. Overall runtime performance for end-to-end tests.
3.0 Mitigations

- Binary instrumentation based
  - Operates on the binary
  - Runtime penalties
- Compiler-based approaches
  - Typically requires access to source code
  - Requires re-compilation phase
- Control flow integrity
  - No source code required
- High rate randomization
  - Could mitigate JIT-rop attack
  - Heavy runtime penalties
4.0 Discussion

- What are the requirements to make CFI a valid mitigation approach to JIT-rop attacks?
5.0 References


