Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming

Dennis Andriesse, Herbert Bos and Asia Slowinska

VU University Amsterdam

DSN 2015
Introduction

Code Integrity Self-Verification on a Hostile Host

- Delay tampering/reversing of software by verifying code integrity
- Application-level: No hardware/kernel support or verification servers
- Prevent malware reversing, cracking, protect critical systems, . . .
Introduction

Code Integrity Self-Verification on a Hostile Host

- Existing work uses checksums $\rightarrow$ broken by Würster et al.
- Oblivious Hashing works, but checks only deterministic program states
- *Parallax* verifies deterministic and non-deterministic paths
Introduction

Return-Oriented Programming

- *Parallax* is based on Return-Oriented Programming (ROP)
- Originally used in exploits to circumvent $W \oplus X$
- Craft ROP programs on stack by chaining returns to *gadgets*
**Protecting Code**

- *Parallax* intentionally creates gadgets to overlap with protected code.
- One or more code regions are translated into ROP verification code.
- Verification code uses the gadgets in the protected code.
- Tampering breaks gadgets $\rightarrow$ verification fails, implicit detection.
- Gadgets can be “unaligned” relative to original instruction stream!
- *Parallax* can be implemented entirely at the binary level.
### Parallax Example

#### Ptrace detector

```
 Parallax Example

Ptrace detector

n+38 <cleanup_and_exit>:
  n+38:  55  push ebp
  n+39:  89 e5  mov ebp,esp
  n+3b:  83 ec 18  sub esp,24
  n+3e:  89 04 24  mov [esp],eax
  n+41:  e8 d5 fe ff ff  call exit@plt

n+46 <check_ptrace>:
  n+46:  55  push ebp
  n+47:  89 e5  mov ebp,esp
  n+49:  83 ec 18  sub esp,24
  n+4c:  c7 44 24 0c 00 00 00 00  mov [esp+0xc],0
  n+54:  c7 44 24 08 00 00 00 00  mov [esp+0x8],0
  n+5c:  c7 44 24 04 00 00 00 00  mov [esp+0x4],0
  n+64:  c7 04 24 00 00 00 00 00  mov [esp],0
  n+6b:  e8 cb fe ff ff  call ptrace@plt
  n+70:  85 c0  test eax,eax
  n+72:  79 07  jns n+7b
  n+74:  b8 01 00 00 00  mov eax,1
  n+79:  eb bd  jmp n+38
  n+7b:  b8 00 00 00 00  mov eax,0
  n+80:  c9  leave
  n+81:  c3  ret
```
Parallax Example

Ptrace detector

n+38 <cleanup_and_exit>:
  n+38: 55                   push ebp
  n+39: 89 e5                mov ebp,esp
  n+3b: 83 ec 18             sub esp,24
  n+3e: 89 04 24             mov [esp],eax
  n+41: e8 d5 fe ff ff       call exit@plt

n+46 <check_ptrace>:
  n+46: 55                   push ebp
  n+47: 89 e5                mov ebp,esp
  n+49: 83 ec 18             sub esp,24
  n+4c: c7 44 24 0c 00 00 00 00 mov [esp+0xc],0
  n+54: c7 44 24 08 00 00 00 00 mov [esp+0x8],0
  n+5c: c7 44 24 04 00 00 00 00 mov [esp+0x4],0
  n+64: c7 04 24 00 00 00 00 00 mov [esp],0
  n+6b: e8 cb fe ff ff       call exit@plt

(gdb) set *(unsigned char*)0x08048479=0x90
(gdb) set *(unsigned char*)0x0804847a=0x90

Dennis Andriesse (VU University)
Parallax Example

Ptrace detector

n+32 <cleanup_and_exit>:
  push ebp
  mov ebp,esp
  sub esp,24
  mov [esp],eax
  call exit@plt

n+46 <check_ptrace>:
  push ebp
  mov ebp,esp
  sub esp,24
  mov [esp+0xc],0
  mov [esp+0x8],0
  mov [esp+0x4],0
  mov [esp],0
  call ptrace@plt
  mov eax,0
  test eax,eax
  jns n+7b
  mov eax,0xc3
  jmp n+32
  mov eax,0
  leave
  ret
Protected Code

Binary Rewriting Rules

- Parallax uses existing gadgets, plus binary rewriting as needed
- Several binary rewriting rules in current prototype:
  - Modify immediate operands, and split instruction to compensate
  - Rearrange code/data to encode (partial) gadgets in offsets
  - Use add for memory operations if mov cannot be encoded
  - Use retf (far return) if a ret cannot be encoded
  - Insert spurious instructions to encode missing gadget prefixes/suffixes
Verification Code

Function-Level Verification
- Select function(s) to use as verification code at binary or source level
- Use modified ROPC compiler to generate verification function
- Verification function uses gadgets used to protect code

Dynamically Generated Function Chains
- Function chains live in data memory → can be generated dynamically
- Enables encryption, self-modification, random selection from equivalent gadgets

Instruction-Level Verification
- Experiments with fine-grained verification code → high overhead due to setup/teardown (2× compared to function-level)
Attack Resistance

Code Restoration Attacks (restore modified code after execution)
- Main threat to any tamperproofing scheme (not applicable in cracking)
- *Parallax* complicates this by choosing verification code that runs often
- Verification code is decoupled from protected code $\rightarrow$ hard to pinpoint

Verification code replacement
- Adversary must craft equivalent code $\rightarrow$ ROP code hard to reverse
- Dynamically generated/self-modifying verification code even stronger

Verification code modification
- Again, adversary must reverse ROP code first
- Verification code is data $\rightarrow$ protectable with (network of) checksums
Evaluation

Coverage and Performance

- *Parallax* protects up to 90% of code bytes with gadget length $\leq 6$, not using spurious instructions (not simultaneously, as rules may conflict)
- Performance overhead $< 4\%$ if verification code outside critical path

Runtime Overhead (Function-Level Verification)

![Graph showing runtime overhead for various applications and techniques]

- clear
- xor crypt
- rc4 crypt
- linear combination

Applications:
- wget
- nginx
- bzip2
- gzip
- gcc
- lame

Total runtime overhead:
- 0\%
- 1\%
- 2\%
- 3\%
- 4\%
Discussion

Dynamic Circumvention

Parallax protects code against explicit modification.
Cannot detect dynamic non-explicit code patching (Pin, DynamoRIO).
Parallax can instead protect specialized detection code for this.

Control-Flow Integrity

Use of ROP requires special consideration when combined with Control-Flow Integrity (CFI).

Protection Coverage (vs Oblivious Hashing)

Parallax protects input-/environment-based code that OH cannot.
Arguably, such code is the most interesting to attackers.
In contrast to OH, Parallax requires no offline testing to compute valid states → can protect even untested/unexplored code.
Conclusion

Summary

- Parallax enables tamperproofing on deterministic and non-deterministic paths, without susceptibility to the attack of Würster et al.
- Up to 90% of code bytes can be protected with gadget length $\leq 6$
- Wisely chosen verification code keeps runtime overhead under 4%
- Performance overhead is in verification code only, isolated from protected code
- Verification code resides in data memory $\rightarrow$ traditional tamperproofing techniques re-enabled for multi-layered protection