Securing Untrusted Code via Compiler-Agnostic Binary Rewriting

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin
Department of Computer Science, The University of Texas at Dallas

Presented by David Gloe

Outline

- Introduction
- Background
- Design
- Implementation
- Evaluation
- Discussion
- Related Work
- Conclusion

Introduction

- Software is often distributed as a binary
- Binaries cannot always be trusted
- Two existing approaches to protection:
  - Virtual Machines (VMs)
  - Binary Rewriting
    - SFI (PittSFIeld, Native Client)
    - CFI (MoCFI)

Virtual Machines

- Pros
  - No need for disassembly
  - Calculate jump targets during runtime
  - Filter API calls with a security policy
  - Damage contained within VM
- Cons
  - Significant Overhead
  - Difficult to formally verify

Binary Rewriting

- Pros
  - No security hardware, software, or VMs needed
  - Better performance than virtual machines
  - Safety can be machine-verified
- Cons
  - Require cooperation from code producers
    - PittSFIeld: gcc-produced assembly
    - Native Client: Use of special compiler
  - Little motivation for producers

Proposed Solution

- REINS: CISC rewriting and in-lining system
  - Binary Rewriting
  - Requires no input from code producers
  - Redirects API calls through a trusted library
  - Jumps are protected by guard code
  - Verifier certifies rewritten binaries are safe

Background

- Assumes Windows, x86, and only binary
- Does not protect code from itself
- Binary must be run at user level
- Defender can modify code before execution
- Statically determining unsafe jump targets is an undecidable problem

System Overview

1) Binary sent through disassembler, generating a control-flow policy
2) Binary rewriting using control-flow policy
3) Rewritten binary verified with trusted verifier
4) Safe binary linked with the policy enforcement library
5) Binary can now be run safely

Design: Rewriting

- Rewriting uses SFI based on PittSFIeld chunks
- Partition into low memory and high memory
- Call instructions placed at the end of chunks
- Jumps referencing Import Address Table are unguarded
- Jump target table used for indirect jumps

Jump Target Table

- Disassembler finds superset of jump targets
- Each old target is replaced with a tagged pointer to its new location
  - Pointers are identified by the illegal hlt opcode (0xF4)
- False positives merely increase binary size
- False negatives are caught by the verifier

Jump Target Table Example

- Assume [r] contains 0xF41234

1) Compare [r] against 0xF4; it matches
2) Move actual address [r+1]=0x1234 into r
3) Sandbox r by ANDing with the bitmask
4) Jump to actual address 0x1234

Code Transformations

`call/jmp r`	`cmp byte ptr [r], 0xF4`
	`cmovz r, [r+1]`
	`and r, (d - c)`
	`call/jmp r`

`ret`	`and [esp], (d - c)` `ret`
`mov rm, [IAT:n]`	`mov rm, offset tramp_n`

`jmp [IAT:n]`	`tramp_n:`
	`and [esp], (d - c)`
	`jmp [IAT:n]`

Design: Memory Safety

- Low memory non-code sections marked as non-executable (NX) by rewriter
- API calls which can unset NX are wrapped
- Untrusted self-modifying code is rejected

Design: Verifier

- Verifier is the only trusted component
  - Executable sections are in low memory
  - Exported symbols target low memory chunks
  - No disassembled instruction crosses a chunk
  - Static branches reference chunks
  - Computed jumps are masked
  - Jumps using the IAT access an IAT entry
  - No trap instructions

Implementation

- Prototype for 32 bit Windows XP/Vista/7/8
- Rewriter
- Verifier
- API Hooking Utility
  - Replaces some IAT entries with trusted funcs
- Intermediary Library
  - Replaces standard kernel32 library

Evaluation

- Median results for COTS applications
  - 100% executable file size increase
  - 41% code size increase
  - 15% process size increase
  - 4.1 seconds binary rewriting time
  - 49 milliseconds verification time
  - 2.4% runtime increase (some decreased)
- Maximum 15% runtime increase

Policy Enforcement Library

- Libraries are automatically created through policy specifications
- Example: disallow sending emails
  - `function conn = ws2_32::connect(SOCKET, struct sockaddr_in *, int) -> int;`
  - `event e1 = conn(_, { sin_port=25}, _) -> 0;`

Case Studies

- Email Client Eureka
  - Prohibits creating executables and executing explorer
- DOSBox Emulator
  - Prohibited access to portions of the file system
- Normal behavior unaffected, policies enforced
- Various Malware
  - All rejected during rewriting or runtime

Discussion

- Does not enforce Control Flow Integrity (CFI)
  - System call policies help
- Assumes jump targets are not dense
  - Each jump target must be one byte more than the word size apart from the next target
  - This is fairly rare
- Relies on classification of code and data
  - Inaccurate classification results in corruption

Classification of Code and Data

- “Differentiating code from data in x86 binaries” by Wartell et. al.
- General disassembly is impossible
  - Reduces to halting problem for x86
- Instruction reference array maps opcodes to instruction lengths
- Utility function estimates likelihood of transition from code to data, or data to code

Related Work

- Source-level SFI
  - Rely on cooperation from code producers
- Binary-level SFI
  - Dynamic approaches have performance issues
  - Static approaches still require code producer input
- System-level
  - Cannot block attacks between modules

Conclusion

- REINS monitors and restricts API calls of untrusted x86 binaries
- Requires no source or debugging information
- Behavior-preserving for many COTS binaries
- Enforcement entirely on user level
- Median runtime overhead of only 2.4%
- Process size increase of only 15%

Questions?

Securing Untrusted Code via Compiler-Agnostic Binary Rewriting

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin
Department of Computer Science, The University of Texas at Dallas

Presented by David Gloe