


# Protecting Against Unexpected System Calls



C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, J. H.  
Hartman  
Department of Computer Science  
University of Arizona

Presented By: Mohamed Hassan

# Code Injection

---

- ❑ What is a system call?
- ❑ What is code Injection?
- ❑ How it works?
  - Exploit a vulnerability in a program (usually a server)
  - Introduces the attacker code into the program
  - Trick the program to run the attacker code
  - To do real damage (get root shell, change permissions,..) the attacker code will execute one or more system calls



# How to defeat it?

---

- ❑ Use a mix of defense mechanism to prevent code injection
- ❑ Defense mechanisms
  - Novice:
    - ❑ Interrupt address table (IAT)
    - ❑ Disguising system call instruction
    - ❑ Code Pocketing
  - Existing ones
    - ❑ Dead and useless code insertion
    - ❑ Randomization and Binary obfuscation



# Some Basics

---

- ❑ System call mechanism
  - Load arguments in registers EBX, ECX, EDX, ESI and EDI (if more push it on the stack)
  - Load syscall number into EAX
  - Execute “int 0x80”
  - HW Pushes PC on the stack (next instruction)
  - Pass control to the kernel
- ❑ ELF (Executable and Linkable format)
  - Extensible Binary file format
  - It contains a Header that describes the subsequent sections that contain code, data, bss and GOT
- ❑ Dynamic Vs Static linking?



# More basics

---

- Static linking
  - Reference to library function are included during compile-time/link-time.
- Dynamic Linking
  - Linked during runtime by the dynamic loader
  - Each routine has unique entries in the GOT (global offset table) and PLT (procedure linkage table)
  - In lazy binding, these entries points to the dynamic linker which will locate the referenced routines (first call only) and update the GOT and PLT tables
  - In eager binding, the linker will resolve all entries during the setup phase of he program

# Attack Model

---

- ❑ The attacker ultimate goal is to execute a system call
- ❑ How?
  - Attacker code invoke the system call directly (direct)
  - Attacker will scan the code for a function that makes system call (mimicry attacks)



# Prevention I

## (Direct System call attack)

---

- Interrupts Address Table (IAT)
  - Add semantic information about the system calls invoked in the program
  - Each system call entry in the IAT of the next instruction and the system call number
  - The IAT is stored in an internal data structure in the kernel (Why?)
  - In dynamic linking, the linker will load the library IAT
- If an attacker calls a system call and the call is not in the table or the next instruction (on the stack) doesn't match any then it is an intrusion

# Prevention II

## (code scanning attack)

---

### □ Disguising System Calls

- Replace system calls with faulting instructions (div by zero)
- Kernel lookup the IAT and if it is a disguise the kernel executes the correct Instruction



### □ Hiding the location of functions that make a system call

- Delete all symbol information that refers to linked functions
  - What about dynamic linking? (it needs the these information)
- Unmap the shared object's symbol table
- Add fake entries in the PLT and GOT

# Prevention II Cont'd

---

- ❑ If the attackers cannot find system call or functions they will scan for byte sequence (fingerprints)
- ❑ To prevent this:
  - Add dead and useless code such as `nop`; `add $0, r`; `push r`; `pop r`;
  - Layout randomization and binary obfuscation
  - Pocketing



# Experiment and Results

---

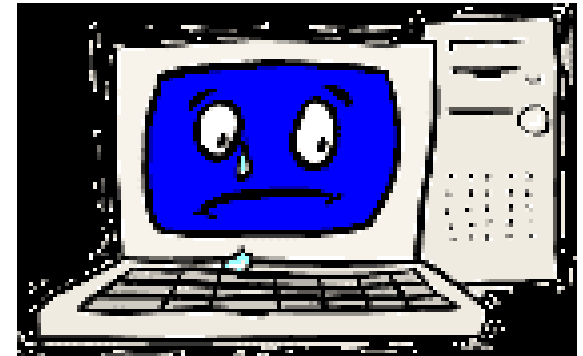
- Target m88ksim (it has a lot of system calls)
- Devised an attack for each case:
  - Known address attack
  - Scanning attack
    - Identify system calls directly
    - Identify functions that will lead to system calls
  - All these synthetic attacks were unsuccessful

# Cost (Overhead)

---

## □ IAT checking overhead on average (SpecInt-2000)

- Runtime/system call: 0.25 usec
- File size: 0.11%
- Memory Size: 0.45%
- Total exec time: 1.7%



## □ Transformation costs (to avoid scanning attacks)

- Randomization and nops :Much smaller than other optimization techniques
- Pocketing: 85% overhead

# Take Away

---

- If you can integrate related ideas together in a reasonable manner, you can end up with a good idea
- Criticism
  - Many, much , few (Do the math)
  - Review the paper before submittingx