

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

JITSec

Just-in-time Security for Code Injection Attacks

Willem De Groef

Nick Nikiforakis, Yves Younan, Frank Piessens
K.U.Leuven, Dept. of Computer Science

willem@cqrit.be

29 november 2010

Who am I?

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- master student @ K.U.Leuven
- strong interests in formal and more applied (or low level) software security
- made it up to here with help of *Nick Nikiforakis*, dr. *Yves Younan* and prof. *Frank Piessens*

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- C/C++ still very popular despite being unsafe
- allow for all kinds of code injection attacks
- many countermeasures over the last 15 years
- three popular and complementing countermeasures
 - ① address space layout randomization (ASLR)
 - ② probabilistic protections (e.g. StackGuard)
 - ③ non-executable heap/stack (e.g. $W\oplus X$ or DEP)

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- most modern OS's ship with a combination
- still, last year has been a showcase of code injection attacks
- countermeasures have their problems:
 - some are fast and efficient
 - but rely on secrecy of memory
 - but limit the use of programming techniques
 - some are really slow
 - some involve heavy binary rewriting

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- Just-In-Time compilation
- hybrid between interpreted and compiled programs
- portability of compiled programs
- speed up of execution time
- JIT is used almost everywhere (Java, .NET, browsers, ...)

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- clash between $W \oplus X$ and JITing:
 - interpreter creates executable code
 - writes to code memory
 - memory cells for that code are not executable anymore!
- developer solution: de-activate $W \oplus X$ for specific memory region
- leaves a hole for attackers to exploit
- real exploitation scenario presented at BlackHat DC 2010 by *Dionysus Blazakis*

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- 1 attacker-controlled variable to inject code
- 2 change memory location to transfer control flow (return address, function pointer)
 - place code of his choice in memory/variable
 - transfer control flow to that code
 - try to gain more control by **issuing system calls from injected code** (on the stack/heap)

Attack scenario

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

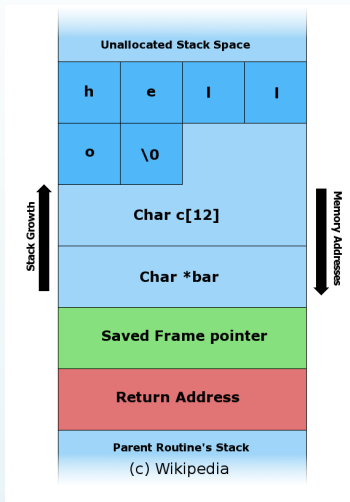
Evaluation

Overhead

Security Policies

Related Work

Conclusions



- shellcode will use system calls
`MOV 0x25, %EAX`
`INT 0x80`
- interrupt in shellcode will trigger kernel
- use 0x80 to locate ISR for system calls
- use 0x25 as offset for system call routine

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- observations:
 - ① $W \oplus X$ is too restrictive
 - ② $W \oplus X$ is used to prevent bad interaction

Research question

Is it possible to build a monitor to mitigate bad system calls without losing the non-executable stack/heap idea?

- define in context of the attacker model
 - mitigate?
 - bad system calls?

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

```
void handle_system_call (int sc)
{
    exec sys_call_table[sc]
}
```

Figure: Normal system

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

```
void handle_system_call (int sc)
{
    exec sys_call_table[jitsec_monitor(sc)]
}

int jitsec_monitor (int sc)
{
    if sc from (heap | stack)
        return EXIT
    else
        return sc
}
```

Figure: System with JITSec

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- EIP points to instruction to execute after INT

...	other instructions
int 0x80	handle the system call
next instruction	← EIP points to here

- to handle INT, save program state in kernel structure
- look up the saved EIP value
- see if it fits in the memory region for stack or heap

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- solution is processor specific (like non-executable stack/heap)
- implemented for the Linux kernel as a loadable kernel module
- all system calls will go through our JITSec monitor

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- inserting some extra code results in extra *CPU* and *memory* usage
- micro benchmarks
 - max overhead $< 12\%$ (upper bound)
- SPEC CPU2000 Integer
 - max overhead $< 5\%$
- conclusion: **average CPU overhead $< 2\%$**

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- memory footprint of applications doesn't change
- extra memory for the kernel is required
- introduction of the monitor itself
 - 180 bytes of kernel code
 - for every system call extra stack space but reclaimed
- conclusion: **extremely low memory overhead**

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- security properties:
 - ① protects all applications
 - ② non-bypassable
 - ③ protects against attacks defined in attacker model
- does not protect against
 - injecting itself
 - return-into-LIBC attacks
 - specific kernel injection attacks

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- lot's of related work
- difference with JITSec lies in the fact that
 - not susceptible to mimicry attacks or false positives
 - no crypto and no memory secrecy
 - no change to software and no sourcecode required
 - can be used in combination with other techniques
 - automatically protects every running application

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

- JITSec fixes a discrepancy between a legitimate programming technique (JIT) and a security countermeasure ($W \oplus X$) by disallowing system calls from stack/heap
- nothing spectacular but
 - ① simple idea
 - ② simple but effective implementation
 - ③ extra line of defense

JITSec

Willem
De Groef

Introduction

Countermeasures

JIT

Problem

Hypothesis

JITSec

Evaluation

Overhead

Security Policies

Related Work

Conclusions

<http://www.cqrit.be/jitsec/>

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy and by the Research Fund K.U.Leuven.