Walls, Gates, and Guards

Thomas Gross

ETH Zurich

Joint work with A. Barresi (xorlab) and M. Payer (Purdue)
In case you want to leave ...

- “Safe” languages, and verification and analysis tools, allow us to build software systems that are shielded from attack
  - Like a walled city – can’t get in
- Programs written in “unsafe” languages are plentiful
  - These languages are useful – like gates in a wall
  - You may not know you are using one
- Until tools easily handle such languages we should use the abundance of computing cycles to protect systems
  - Guards and standing armies made cities secure and functioning
- Dynamic control-flow integrity is a promising technique to detect attacks that exploit memory errors
Memory-safe languages

- Automatic memory management
- All address arithmetic hidden from user
  - No buffer overflows, out-of-bounds array accesses, arbitrary type conversions, ...
- Restrict memory space that can be accessed by user program
- Either by language design or by static code analysis

Hope: a weapon against memory errors
- Memory error: any corruption of memory
Memory errors & vulnerabilities

Come in various forms ...

- Allow attackers to corrupt memory in a more or less controllable way
- Problem: modification of arbitrary memory location
  - Worst case: attackers gain right to execute arbitrary code
- Exist in programs written in “unsafe“ languages that do not enforce memory safety
Safe languages

- Just use a memory-safe language?
Safe languages

- Just use a memory-safe language?
  - Popular memory-safe languages based on a virtual machine (VM)
  - "Language VM", e.g., JVM
    - Provides framework for access control
    - Provides environment for multi-tier compilation (performance)
Safe languages

- Just use a memory-safe language?
  - Popular memory-safe languages based on a virtual machine (VM)

- But “language VM”
  - May be implemented in an unsafe language
  - May use or provide interface to unsafe libraries

- Memory errors are still an issue
Attacking safe language VMs

- **Example: Java VM**
  - CVE-2013-1491
  - Target: Oracle Java SE 7 / 6 / 5
  - Memory error in OpenType fonts handling within native layer of JRE
    - Leveraged to arbitrary code execution
    - Completely bypassed state-of-the-art defenses
      (DEP & ASLR – later more)

Demonstrated at Pwn2Own at CanSecWest 2013 by Joshua Drake (on Windows 8 + Java SE 7 Update 17)
http://www.accuvant.com/blog/pwn2own-2013-java-7-se-memory-corruption
Memory errors still an issue

- Language VMs for “safe” languages implemented in “unsafe” language
- “Unsafe” languages like C/C++ are still very popular
  - Prediction: C/C++ will be with us for a long time
  - Yes, there are alternatives ..... sometimes
  - Yes, the list of alternatives is growing ... for some situations
- So we should take a look

http://www.langpop.com/
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
Memory errors

- Old problem: modification of arbitrary memory location

- Memory errors can lead to serious security vulnerabilities
  - Worst case: attackers gain arbitrary code execution capabilities
Common vulnerabilities and exposures (CVE)

- High Severity
  - Memory Errors: 36.29%
  - XSS + CSRF: 0.1%
  - SQL Injection: 19.84%

- Medium Severity
  - Memory Errors: 8.05%
  - XSS + CSRF: 28.89%
  - SQL Injection: 2.75%

- Low Severity
  - Memory Errors: 3.34%
  - XSS + CSRF: 0%
  - SQL Injection: 31.39%


Memory Errors: CWE-119, CWE-399 "use after free", CWE-189 in High / XSS + CSRF: CWE-79, CWE-352 / SQL Injection: CWE-89

https://cve.mitre.org/
Modern software stack

- Application
- Web Browser
- Client-side Script
- Java Application & Libraries
- Libraries & Tools
- Operating System
- Hardware
Modern software stack

Potentially prone to memory errors & corruption
Java VM written in C/C++
Safe languages (VM based)

- Attacker may exploit memory errors
  - In the VM
  - In unsafe libraries used by VM or application
Java VM written in C/C++

Software Stack

Java Application Process

Java Application

Java API & Libraries

Execution Engine – JIT | GC

Java VM

Libraries

Operating System

Hardware

Potentially prone to memory errors & corruption
“Unsafe“ languages

- Allow low-level access to memory
  - Typed pointers & pointer arithmetic
  - No automatic bounds checking or index checking

- Weakly enforce typing
  - Cast (almost) anything to pointers

- Explicit memory management
  - Like malloc() & free() in C
Types of memory errors

- **Spatial error**
  - De-reference pointer that is out of bounds
  - Read or Write operation

- **Temporal error**
  - De-reference pointer to freed memory
  - Read operation
Exploiting memory errors

- **Spatial error**
  - `*ptr` ➔
  - Array / Object
  - Attacker Supplied Data
    - Overwrites/Reads Data/Pointers
  - Overwrite data or pointers
  - Used or de-referenced later

- **Temporal error**
  - `*ptr` ➔
  - Attacker Supplied Data
    - Used as Wrong Type
  - Make application allocate memory in the freed area
  - Used as old type
Attackers use memory errors to

- **Overwrite data or pointers**
  - Function pointers, sensitive data, index values, etc.

- **Mislead information**
  - E.g., corrupt a length field

- **Construct attacker primitives**
  - Write primitive (write any value to arbitrary address)
  - Read primitive (read from any address)
Attack types

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

Attack model according to: „sok: eternal war in memory“ laszlo szekeres, mathias payer, tao wei, dawn song
Attack types

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

Attack model according to: „sok: eternal war in memory“ laszlo szekeres, mathias payer, tao wei, dawn song
Control-flow hijack attacks

- Most powerful attack

- Hijack control-flow
  - To attacker-supplied arbitrary machine code
  - To existing code (code-reuse attack)

- Corrupt code pointers
  - Return addresses, function pointers, vtable entries, exception handlers, jmp_bufs
Control-flow hijack attacks

- Most ISAs support indirect branch instructions
  - E.g., x86 “ret”, indirect “jmp”, indirect “call”

- fptr is a value in memory at 0xafe08044
  - branch *fptr

```
Code

0x08056b30

good_func:

0x8056b30

fptr: 0xafe08044

fptr
0x08056b30
```
Control-flow hijack attacks

- fptr is a value in memory at 0xafe08044
  - branch *fptr
  - fptr was corrupted by an attacker

- Attacker goal: hijack control-flow to injected machine code or to “evil functions”
Control-flow hijack to injected code

[Diagram showing the control-flow hijack to injected code with stack, heap, and code sections.]

Indirect call to func()  Hijacked indirect call
State of the art defenses

- Non-executable data
  - NX bit

- Data Execution Prevention (DEP)
  - OS support
Non-eXecutable data (NX)

- Make data regions non-executable (by default)

- Changing protection flags or allocating rwx memory still possible (on most systems)
  - Required for JITs
NX / DEP

- Binary images need to provide separate sections/segments that can be mapped exclusively as rw- OR r-x
  - Linker support required

- Self-modifying code not allowed
  - Compiler support required
    - If code is generated just-in-time, explicit rwx allocation required
Bypassing NX / DEP

- Only use existing code
- Code-reuse attack
  - ret2libc, ret2bin, ret2* attacks
  - Return-oriented programming (ROP)
  - Jump/Call-oriented programming
- Use code-reuse technique to change protection flags
  - Allocate or make memory executable
    - mprotect/VirtualProtect
    - mmap/VirtualAlloc
Return-oriented programming (ROP)

- Use available code snippets ending with `ret` instruction
  - Called gadgets or ROP chain
  - E.g., write primitive

```
1 pop %edx; ret;
2 pop %eax; pop %ebx; ret;
3 mov %edx, (%eax);
    mov $0x0, %eax;
    ret;
```

Stack

Code
Return-oriented programming

- Very powerful!
  - Turing complete although not required

- Need to be in control of memory `%esp` is pointing to
  - Or make `%esp` point to area under control

- Also possible with `jmp` or `call` gadgets
  - Complicated to keep control and dispatch to the next gadget
  - Generalization: Gadget-Oriented Programming
Addresses in memory

- To hijack control-flow or to corrupt memory an attacker needs to know where things are in memory
  - Addresses of data or pointers to corrupt
  - Addresses of injected code (shellcode)
  - Addresses of gadgets

- Sometimes it's enough to know the rough location but most of the time attackers need the exact location
  - Corrupting only least significant bytes i.e. an offset might work in some special cases (but not in general)
ASLR

Today most operating systems implement *Address Space Layout Randomization* (ASLR)

What can be randomized?

- OS: Stack, heap and memory mapping base addresses
- OS, compiler, linker: Executables and libraries
  - Position-independent or relocatable code
Bypassing ASLR

- **Low entropy**
  - Brute-force addresses
    (multiple attempts required)

- **Memory leaks (information disclosure)**
  - Leak addresses to derive base addresses
    - E.g., run-time address pointing into a library
    - Construct and enforce a leak by memory corruption

- **Application and vulnerability specific attacks**
Memory leak

```
0x1|0x2|0x4
4096
address shellcode
address mprotect()
dummy ebp
buf[1024]
Attacker Code
"Shellcode"
```

```
Stack
0x0ef4604

Heap
libc data
libc
vulnExecutable
```

```
mprotect = leaked pointer – static offset
0x0ebe0880 = 0x0e4604 - 0x003f3d84
```
Generic defense: DEP & ASLR

- **DEP**: Data Execution Protection
- **ASLR**: Address Space Layout Randomization

- Exploitation becomes harder for all vulnerability classes & attack techniques

- Together quite effective
  - If implemented correctly and used continuously

- But DEP and ASLR not enough
Compile-time protection

- Usually require source code changes (annotations) and/or recompilation of the application
  - To add run-time checks

- Stack canaries / Cookies
- Pointer obfuscation
- /GS (buffer security check)
- /SAFESEH (link-time, provide list of valid handlers)
- SEHOP (run-time, walk down SEH chain to final handler before dispatching / integrity check)
- Virtual Table Verification (VTV) & vtguard
- Control-Flow Guard (new in Visual Studio 2015)
Stack canary / cookie

```c
void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}
```
void vulnFunc() {
    <copy canary>
    char buf[1024];
    read(STDIN, buf, 2048);
    <verify canary>
}

Stack at function exit

overwritten frame
overwritten retaddr
overwritten ebp
overwritten canary
buf[1024]

Stack during vulnFunc()

main() stack frame
return address
saved ebp
stack canary
buf[1024]

copy canary
%esp →
verify canary
%ebp →
stack canary
%esp →
stack canary
%ebp →
Stack canary / cookie

- Detects linear buffer overflows on stack
  - At function exit

- Corruption of local stack not detected
  - Only if canary / cookie value is overwritten

- Incurs runtime overhead

- Effectiveness relies on secret
  - Leaking, predicting, guessing or brute-forcing might work in special cases
DEP & ASLR

- DEP & ASLR are not enough

- A determined attacker will use code-reuse techniques and memory leaks to bypass DEP & ASLR
  - And application specific bypasses/properties
More defenses

- DEP and ASLR based on memory model
  - Prevent/complicate attacker access to memory

- Programs execute instructions
  - More involved than use of memory

- Goal: protect program *execution*
Attacker model

- Let's assume a powerful attacker
  - Can arbitrarily corrupt data and pointers
  - Can read entire address space of a process

- Only restriction on attacker:
  - No data execution and no code corruption (NX/DEP/W^X)
Question

- Can we still prevent arbitrary code execution and code-reuse attacks?
Observations

- Attacker needs to hijack control-flow
  - To injected or existing code

- VM/runtime system must ensure that control-flow stays on the intended legitimate path
  - As allowed by compiler resp. control-flow graph (CFG)
Control-flow integrity (CFI)

- Construct a control-flow graph (CFG)
  - Should be as strict as possible

- Ensure that control-flow stays within CFG
Control-flow integrity (CFI)

- Original publication in 2005
  - “Control-Flow Integrity – Principles, Implementations, and Applications“
  - M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti
  - CCS'05 (ACM Trans. on Information and System Security (TISSEC) 13(1) Oct 2009)

- Many CFI implementations were proposed during recent years
  - Compiler-based
  - Binary-only (static rewriting)
Control-flow integrity (CFI)

- Construct a control-flow graph (CFG)
  - Should be as strict as possible

- Ensure that control-flow stays within CFG

- If no path within the CFG can be misused by an attacker then the CFI policy can be considered secure
Control-Flow Integrity (CFI)

- Basic block
- Direct branch
- Indirect branch
Hijacked control-flow

- Basic block
- Direct branch
- Indirect branch

Nodes represent basic blocks, with direct branches indicated by solid lines and indirect branches by dotted lines. The 'ret' node is highlighted, indicating a return point in the control-flow graph.
Control-Flow Integrity (CFI)

- Basic block
- Direct branch
- Indirect branch
Control-Flow Integrity (CFI)
Control-Flow Integrity (CFI)
Control-Flow Integrity (CFI)

- Basic block
- Direct branch
- Indirect branch under CFI

CFI VIOLATION
Control-flow integrity (CFI)

- **Drawbacks of proposed solutions**
  - Too permissive CFG due to over-approximation
  - Need to recompile
  - No support for shared libraries

- **Most solutions shown to be ineffective**
  - “Hardened” exploits still worked under CFI
Control-flow integrity (CFI)

- Static CFI not enough: Dynamic approach necessary
  - Dynamic CFI
Lockdown – dynamic CFI

- Enforces a strict CFI policy for binaries
- Supports shared libraries & dynamic loading
- Constructs and enforces CFG at runtime
  - Using static and dynamic information
Lockdown – dynamic CFI

Lockdown – design

- Dynamic binary translation to instrument code with additional CFT checks
  - Basically a user-space VM
  - Ensures no untranslated code is ever executed

- A trusted loader loads ELF dynamic shared objects (DSOs) and provides symbol information for CFG construction

Lockdown – design

- Separation of domains achieved by
  - Separate memory areas
  - Randomization of locations
  - Trampolines
  - Information leak prevention

- Stronger guarantees achieved by marking Lockdown areas as read-only during code-cache execution
Lockdown – attacker model

- Like in general CFI Attacker Model
  - Can arbitrarily corrupt data and pointers in application domain
  - Can read entire address space of application domain
  
- Only restriction on attacker
  - No data execution and no code corruption (NX/DEP/W^X)
Lockdown – High-Level CFI policy

- **call policy**
  - Allow calls to imported & exported symbols
  - Allow calls to local symbols

- **jmp policy**
  - Allow local jumps within symbol boundaries
  - Allow jumps to local symbols

- **ret policy**
  - Shadow stack (allows reauthentication)
Lockdown – CFI policy for calls

/bin/<exec>

/lib/libc.so.6

/lib/lib*

symbol table of ELF DSO
.text section of DSO

allowed control flow transfer
illegal control flow transfer
Lockdown – CFI policy for returns

- **Instrument calls and returns**
  - Return address pushed to a shadow stack
  - Upon return: return address is compared to value on shadow stack
    - Resynchronization possible
  - If values don't match raise exception
Lockdown – CFI policy for returns

- call instrumented such that
  - Return address is pushed onto the shadow stack and the application stack
  - Control-flow is transferred to callee

Stack at call

Shadow Stack

Application Domain

Lockdown Domain
Lockdown – CFI policy for returns

- call instrumented such that
  - Return address is pushed onto the shadow stack and the application stack
  - Control-flow is transferred to callee

![Stack diagram]

- Stack after call
  - caller stack frame
  - return address 6
  - saved ebp
  - local vars

- Shadow Stack
  - return address 3
  - return address 4
  - return address 5
  - return address 6

Application Domain

Lockdown Domain
Lockdown – CFI policy for returns

- `ret` instrumented such that
  - Return address on the application stack is compared to value on shadow stack
  - If values differ, try to resynchronize else raise exception

---

**Stack at ret**
- `%ebp` → caller stack frame
  - return address 6
  - saved ebp
  - local vars
- `%esp` → Application Domain

**Shadow Stack**
- return address 3
- return address 4
- return address 5
- return address 6
- Lockdown Domain

---

**Diagram Notes**
- `==` indicates comparison
- `top` indicates stack top
Lockdown – challenges

- Detection of callbacks & function pointers
  - No information regarding types at runtime
  - If stripped, no extended symbol information
    - Coarser-grained CFG

- Control-flow transfers do not always adhere to the rules presented

- Overhead of CFT checks
Lockdown – implementation

- **Heuristics for function pointer detection**
  - E.g., “leal imm32(%ebx), %e*%x”
  - E.g., relocation entries like R_386_RELATIVE

- **Special handling of control-flow specifics**
  - E.g., PLT inlining
  - E.g., whitelisting of runtime support CFT

- **Several inlined performance optimizations**
Preliminary performance evaluation

- SPEC CPU2006
- 29 programs
- Total 27 benchmarks
  - 2 benchmarks missing
  - Tool chain problems
Lockdown – good and bad performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BT overhead</th>
<th>Lockdown overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>108.85%</td>
<td>148.16%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>6.65%</td>
<td>6.79%</td>
</tr>
<tr>
<td>403.gcc</td>
<td>41.67%</td>
<td>52.22%</td>
</tr>
<tr>
<td>433.milc</td>
<td>4.05%</td>
<td>7.92%</td>
</tr>
<tr>
<td>444.namd</td>
<td>1.73%</td>
<td>2.08%</td>
</tr>
</tbody>
</table>

Intel Core i7 CPU 920@2.67 GHz with 12GiB Ubuntu Linux 12.04.4 LTS 32-bit x86 / gcc 4.6.3

- **Avg overhead Lockdown:** 19.09%
  - Overhead binary translation alone: 14.64
- **Most benchmarks overhead below 20%**
  - Only 5 benchmarks over 45%
Lockdown – security evaluation

- Unfortunately most static CFI solutions were shown to be ineffective

- (D)AIR bad in measuring CFI security effectiveness
  - LibreOffice has 56'417'429 bytes of executable memory
  - 99% (D)AIR allows 1% of the bytes as attacker targets
    - 564'174 potential targets
  - Attacker normally just needs a handful of gadgets to mount a successful code-reuse attack
Dynamic CFI

- Key idea: use binary translator to rewrite program on the fly
  - Practical solution
- Works for arbitrary x86 binaries
  - No source code needed
- Binary translator adds overhead
  - Less than 15% for many programs
Dynamic CFI

- Key idea: use binary translator to rewrite program on the fly
  - Practical solution
- Works for arbitrary x86 binaries
  - No source code needed
- Binary translator adds overhead
  - Less than 15% for many programs
- Binary translator with dynamic CFI guards against (some) attacks
  - No complete protection
Lockdown – dynamic CFI

CFT: Control-Flow Transfer, ICF: Indirect Control-Flow,
ELF: Executable and Linkable Format, DSO: Dynamic Shared Object
Thanks to

- Antonio Barresi
  - Now at xorlab
- Mathias Ganz
  - Now at xorlab
- Mathias Payer
  - Now at Purdue
Concluding remarks

- Control-flow integrity protects program execution paths
  - Static CFI elegant but not practical
- Dynamic CFI offers chance to block wide avenue
  - More work needed
  - Implementation
  - Evaluation models
- Spend cycles on guarding execution of programs
  - No (user) program should run on bare hardware
  - A layer of indirection adds overhead – but protects
Thank you for your attention