RAP: RIP ROP

PaX Team

H2HC 2015.10.24

Introduction

PaX/grsecurity

Return Address Protection

Indirect Control Transfer Protection

Overview

- Host Intrusion Prevention System
- 15 years: 2000-2015, linux 2.2-4.2
- Focus: exploitation of memory corruption bugs
- Threat model: arbitrary read-write memory access
- Bugs vs. Exploits vs. Exploit techniques
- Privilege abuse vs. Privilege escalation
- Performance vs. Usability

Memory Corruption Bugs

- Unintended control over address/content of memory access
  - "Precursor" bugs included (memory disclosure, unintended reads, etc)
- Two generic goals:
  - Find them in the source
  - Catch them before they trigger
- Too many kinds to cover them with universal approaches
- see `http://cwe.mitre.org/`

Threat Model

- Union of the powers (unintended sideeffects) of all possible memory corruption bugs
- Arbitrary read-write memory access
- Bug can be triggered:
  1. for arbitrary addresses
  2. with arbitrary content
  3. arbitrary operation
  4. arbitrary number of times
  5. at arbitrary times
- Privilege abuse: exercise existing powers for unintended purposes
- Privilege escalation: gain new powers (to subsequently abuse them)

Exploit Techniques & Defenses

Execute new (injected) code (shellcode)	Non-executable pages, runtime code generation control, ASLR
Execute existing code out-of-(intended)-order (return-to-libc, ROP/JOP)	Control flow integrity, RAP, ASLR
Execute existing code in-(intended)-order (data-only attacks)	Open question (RANDSTRUCT, KERNSEAL, etc)

- Increasing order of difficulty
- Decreasing amount of control

Exploit Techniques vs. Memory Corruption Bugs

Abuse/Escalation	Shellcode	Code Reuse	Data-only
CVE-xxxx-xxxx	✔	✔	✔
CVE-xxxx-xxxx		✔	
CVE-xxxx-xxxx			✔
0-day #1	✔		
0-day #2	✔	✔	
...			

- PaX: started as a defense mechanism for the first column
- Lately: groups of rows (bug classes via gcc plugins)
- Today: second column

Code Reuse Attacks and Defenses

- Code pointer modification (unintended CFG edge)
  - Return addresses
  - Language level function pointers
    - Virtual method table
    - Signal handlers and signal return contexts
    - Exception handling, setjmp/longjmp, landing pads
- Defenses (PaX future doc from 2003)
  - Immutable (read-only) code pointers
    - .rodata, RELRO, CONSTIFY, CPI/CPS
  - Code pointer target verification
    - CFI, RAP
  - Limited performance budget as usual

StackGuard 1997

- Developed in 1997, published in January 1998 at the USENIX Security Symposium
- Crispin Cowan and others
- No formal threat model, mixes up bug category (buffer overflow) with exploit techniques (code injection)
- Probabilistic defense
- Canary between saved return address and the rest of the stack frame
  - Canary: terminator, random
  - No protection for other state (frame pointer, local variables, arguments)

StackGuard 1999 (XOR canary)

- Attack by Mariusz Woloszyn (emsi) in PHRACK 56 in May 2000
  - Abuses unprotected local pointer variable
- Response: encrypt/decrypt return address by a random key
  - Credited to Aaron Grier
  - Mentioned in US7752459 (PointGuard patent)
- Released in StackGuard 1.21 in November 1999, abandoned later without explanation

Stack Shield 1999

- Released in August 1999 by Vendicator
- No formal threat model
- Shadow stack and range checking for return addresses
  - Deterministic defense
- No attack detection originally, added in v0.5
- Limited function pointer protection (range checking) in v0.6

Propolice/Stack Smashing Protector (SSP) 2000

- Developed by Hiroaki Etoh (IBM Japan) and announced on the gcc and bugtraq lists in August 2000
  - Patented (lapsed):US6941473
- No formal threat model
- Probabilistic defense
- Protects the frame pointer, some local variables, arguments
- Served as basis for Microsoft's /GS and Red Hat's reimplementation for gcc 4.1 (February 2006)
- -fstack-protector vs. -fstack-protector-all vs. -fstack-protector-strong (gcc 4.8)

Overview

- Threat model: arbitrary read-write access
  - Except the 'at arbitrary times' part
- Conceptually based on the XOR canary approach
  - Probabilistic defense
  - Performance tweaks without sacrificing security
  - Main pass is in GIMPLE, not RTL
- Mostly architecture independent
  - Single callee saved reserved register (RAP cookie), (`r12` on amd64)
    - Hard to leak (uninstrumented asm code)

RAP Example

```
push %rbx
mov 8(%rsp),%rbx
xor %r12,%rbx
...
xor %r12,%rbx
cmp %rbx,8(%rsp)
jnz .error
pop %rbx
retn
.error:
ud2
```

RAP (kernel)

- RAP cookie changes:
  - Per task
  - Per system call
  - Per iteration in selected infinite loops
    - Long running event handlers (idle loop, kthreadd, etc)
    - Could be automated perhaps
- Unreadable kernel stacks
  - Prevents cross-task infoleaks and corruption
  - Needs per-cpu pgd (already developed for KERNEXEC/UDEREF)
  - Implementation problem: waitqueue structs
    - Move them off the kernel stack

RAP (userland)

- XOR canary method is vulnerable to arbitrary reads
  - ASLR helps a bit: two leaks are needed
    - encrypted return address
    - plaintext code address (not necessarily the return address)
- Combine with return place (code pointer target) verification

Performance Optimizations

- Reduce coverage without sacrificing security
  - Unlike ssp and ssp-strong
- Compute 'can corrupt memory' property for each function and basic block
  - Propagate it up the call hierarchy
    - Omit instrumentation if the function cannot corrupt memory
    - About 9% of kernel functions untouched, 15% in chromium
  - Basic block coverage narrowing
    - Up to 40% of kernel functions, 8% in chromium
- XOR elimination
  - If the RAP cookie does not spill to memory (basic block narrowing can help)
  - About 6% of kernel functions

Overview

- Deterministic defense
- PaX future doc 2003
- CFI at CCS 2005
- Many more variants since (IFCC, VTV)
- Coarse-grained vs. fine-grained
- Fine grained approaches based on the Indirect Control Flow Graph
  - Hard to construct (undecidable in the general case)
  - Approximations
    - RAP: Type-Based Self-Assembling Indirect Control Flow Graph

Fine-Grained Control Flow Integrity

- Constructing the ICFG is hard
  - Vertices are easy
  - Edges not so much
    - Missing edges result in false positives
    - Extra edges result in reduced security
  - Access to entire code
    - Dynamically loaded or generated code
    - LTO, load time/runtime construction
- Vertex categorization
  - Equivalence sets based on the ICFG
  - Argument count
  - Type based

Overview

- Idea: construct the ICFG vertex categorization and have the ICFG approximation emerge automatically
  - Overapproximate the ICFG as much as the language rules allow
  - Based on function and function pointer types
  - False edges possible if unintended functions have the same type
    - Indirect call elimination (devirtualization, etc)
    - Type diversification
- Extract type information for each function and function pointer
- Compute hash based on parts of the type
  - Language construct dependent
- Verify matching hash value between function and function pointer dereference (indirect call, function return, etc)

Type Hash

- C functions and function pointers
- C++ functions and function pointers
  - Non-class functions
  - Static class member functions
  - Non-virtual class member functions
  - Virtual class member functions
    - Ancestor method that every other method overrides
- Type parts
  - Return type
  - Function name
  - Function parameters
    - 'this' parameter for non-static class member functions

Type Hash Parts

Usable parts in type hash	Return	Name	'this'	Parameters
non-class or static member function/ptr	Y	N	N/A	Y
non-virtual method/ptr	Y	N	N	Y
virtual method/ptr	N	N	N	Y
ancestor method/virtual method call	Y	Y	Y	Y

- C++ virtual method return types can be covariant
- C++ method pointers can target both virtual and non-virtual methods
- Ancestor method type can supplant all overriding method types in virtual calls
- Compiler internal representation or source language text

Type Hash Details

- Any hash function will do
  - Initial state allows for easy binary diversification/watermarking
  - Reduce output to desired size (e.g., 32 bits)
- Hash value range assignment:
  - Positive numbers for functions and function pointers
  - Negative numbers for function returns and return places
  - Reserved value for functions whose address is not taken
  - Reserved values for exception handling (setjmp/longjmp/landing pads)
- Store as 64 bits (full sign extended 32 bit value)
- Verify as sign extended 32 bit value

Indirect function call examples

```
cmpq $0x11223344,-8(%rax)
jne .error
call *%rax
...
cmpq $0x55667788,-16(%rax)
jne .error
call *%rax
...
dq 0x55667788,0x11223344
func:
```

Function return example

```
call ...
jmp 1f
dq 0xffffffffaabbccdd
1:
...
mov %(rsp),%rcx
cmpq $0xaabbccdd,2(%rcx)
jne .error
retn
```

Compatibility

- Observation: everyone gets function pointer casts wrong :)
  - gcc(!), glibc, linux, chromium, etc
- Basic rule: function type must match function pointer type used in the indirect call
  - In-between casts to other function pointer types are allowed
    - But almost everyone fails to convert back for the actual indirect call
- Fixing these problems can uncover real bugs
  - Two birds with one stone: might as well do type diversification while at it
- Call to arms to fix all of them :)
  - RAP related fixes

Compatibility examples

```
int (*fptr)(void *, int);
int func1(void *p, int i);
int func2(void *p);
char func3(void *p, int i);
int func4(struct1 *p, int i);
fptr = &func1; //correct
fptr = &func2; //wrong
fptr = &func3; //wrong
fptr = &func4; //wrong
```

Performance

- Linux: both return address and function pointer protection
  - Less than 5% on 'du -s'
  - Over 25% for ssp-all
- xalancbmk: used in SPECCPU but this was the latest xalan-c/xerces-c available in gentoo
  - All dependencies with return address protection only (cf. compatibility note before)
  - Less than 4% on the SPEC reference test file
- chromium:
  - All dependencies with return address protection only (cf. compatibility note before)
  - Less than 8% on dromaeo javascript tests (big variations, some even better than baseline)

Chromium 47.0.2526.16 Statistics

unique	non-virtual	virtual	unique	non-virtual	virtual
functions	235196	159427	indirect calls	10811	101552
hashes	70498	44265	hashes	6467	29567
ratio (func/hash)	3.4	3.6	ratio (call/hash)	1.7	3.4

Chromium 47.0.2526.16 Top Ancestors

```
4050 base::internal::BindStateBase<Diversifier>::~BindStateBase() [with Diversifier = void()]
1976 virtual base::Pickle::~Pickle()
 867 virtual blink::ScriptWrappable::~ScriptWrappable()
 829 virtual ExtensionFunction::~ExtensionFunction()
 707 virtual const blink::WrapperTypeInfo* blink::ScriptWrappable::wrapperTypeInfo() const
 662 virtual google::protobuf::MessageLite::~MessageLite()
 660 virtual int google::protobuf::MessageLite::GetCachedSize() const
 660 virtual google::protobuf::MessageLite* google::protobuf::MessageLite::New() const
 660 virtual void
google::protobuf::MessageLite::SerializeWithCachedSizes(google::protobuf::io::CodedOutputStream*)
const
 660 virtual bool
google::protobuf::MessageLite::MergePartialFromCodedStream(google::protobuf::io::CodedInputStream*)
 660 virtual int google::protobuf::MessageLite::ByteSize() const
 660 virtual void google::protobuf::MessageLite::Clear()
 660 virtual bool google::protobuf::MessageLite::IsInitialized() const
 637 virtual std::string google::protobuf::MessageLite::GetTypeName() const
 637 virtual void google::protobuf::MessageLite::CheckTypeAndMergeFrom(const
google::protobuf::MessageLite&)
 501 virtual ui::LayerDelegate::~LayerDelegate()
```

Indirect Function Call Conversion (Devirtualization)

- Observation: some hash values get assigned to a single virtual method in a program
  - All virtual method calls with the same hash can only resolve to this one method
    - Devirtualization opportunity missed by normal approaches
- Works for non-virtual methods and pointers too

Summary

- RAP provides comprehensive indirect control flow protection
- Function returns
  - Return address encryption (XOR canary): probabilistic, precise
  - Return place type checking: deterministic, approximation
- Indirect calls
  - Call target type checking: deterministic, approximation
- Very low performance impact
- Scales to real world software

http://pax.grsecurity.net

http://grsecurity.net

irc.oftc.net #pax #grsecurity