Synthetic Memory Protections

An update on ROP mitigations

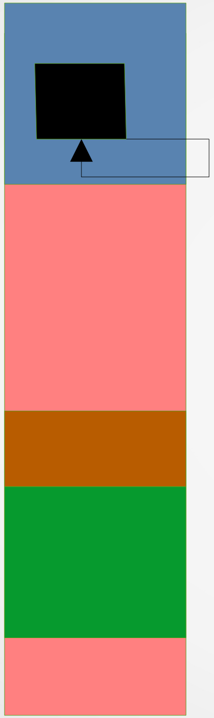
Theo de Raadt
OpenBSD

Attack methods advance

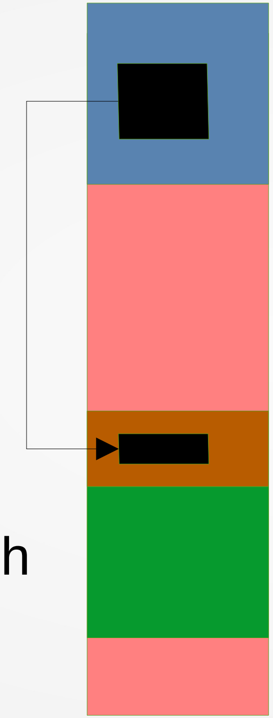
- Smashing the Stack, 1996
  - Solution: make the stack non-executable, 1999?
- Payload on heap, 1998
  - Solution: make the heap non-executable, 2001+?
- Then came ROP. A stack payload contain sequential ret's to pre-existing code chunks (called gadgets) already present in code memory, combining them however it takes to gain control
  - ASLR and other mechanisms to hide code locations
  - But info leaks can disclose code locations…
  - There isn't a simple complete solution to block ROP.

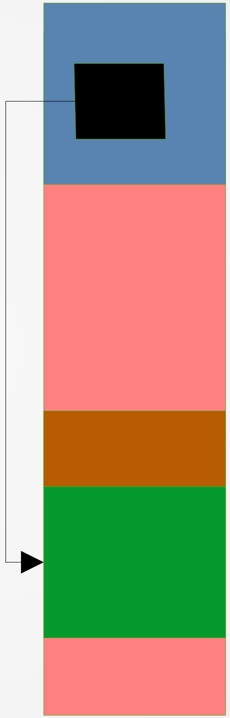
Attack methods

Smash
The
Stack



Smash
The
Heap





Return
Oriented
Programming

- point at many
  pieces of code

Code must remain executable so how do we stop ROP?

So the solutions for ROP are incomplete

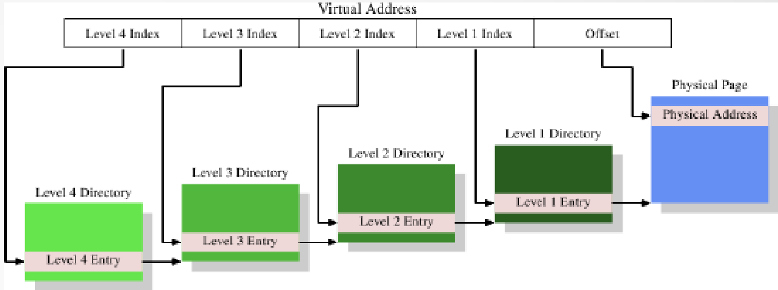
- ROP methods have become increasingly sophisticated
- But we can identify system behaviours which only ROP code requires
- We can contrast this to what Regular Control Flow code needs
- And then, find behaviours to block

25 years of stack smashing mitigations

- 1st generation: non-X stack, W^X, and stack protector
- 2nd generation: ASLR and other hiding methods
- 3rd generation: RETGUARD and gadget reduction
  (Todd Mortimer RETGUARD Tokyo)
- 4th generation: Synthetic Permissions

Natural Abilities of the MMU

- Remap physical memory into virtual ranges
- Generally two virtual ranges:
  - Kernel
  - Userland (* focus of talk)



- Various approaches, all with same basic idea:
  - Tree structure, hardware/software walked, cached in a TLB
  - Entries contain Physical Page address, plus Attribute bits
  - Attributes bits include Permission bits: R, W, etc

R and W

- Older MMU only had 2 Permission bits
  - Present meaning Valid
  - Write

- Valid implies Read
- Read implies either program reading memory or instruction fetch
- (Instruction fetch is also known as X)
- Better MMU had Valid bit, and seperate R and W

- Permission set: no mapping, read+execute, read+write+execute.

X or NX

- Around 1999, newer cpus added an X permission

- But some added Not-eXecutable, or NX, instead
- Confusing. Due to V = Read, so for software compatibility the inverse permission added as NX
- Operating systems had to support old and new systems..

- OpenBSD was first system to use X/NX on all possible platforms with a policy called W^X (which was a solid step in 2002...)

Introducing new Synthetic Permissions

- Immutable mappings
- Execute-Only, in hardware where possible, but also:
  - Opportunistically block Read before Execute
  - Block System Calls from reading userland memory
- Stack Permission on mappings
- Syscall Permissions on mappings
- Pinning Syscall entry to a unique entry point

Procmap tool shows new permissions

- From the OpenBSD manual page

```
# procmap -a
Start     End              Size  Offset    rwxSeIpc  RWX  I/W/A ...
08048000-080b0fff          420k 00000000 r-x---p+ (rwx) 1/0/0 ...
...
```

In this format the column labeled “rwxSeIpc” comprises:

rwx	permissions for the mapping
S	mapping is marked stack
e	mapping is allowed system call entry points
I	mapping is immutable (rwx protection may not be changed)
p	shared/private flag
c	mapping needs to be copied on write (‘+’) or has already been copied (‘-’)

Procmap of sed(1)

- Sample output (edited)
- Removed most malloc(3)
- Notice:
  - Random layout
  - X without R
  - Many I (Immutable)
  - Some e (Syscall)
  - Unmapped guards
  - S (Stack) near end

```
Start          End                Size Offset   rwxSeIpc  RWX   Object
0e1330a60000-0e1330a62fff         12k 00000000 r----Ip+ (rwx) sed rodata
0e1330a63000-0e1330a68fff         24k 00002000 --x--Ip+ (rwx) sed text
0e1330a69000-0e1330a69fff          4k 00000000 r----Ip- (rwx) sed relro
0e1330a6a000-0e1330a6afff          4k 00000000 rw---Ip- (rwx) sed data
0e1330a6b000-0e1330a6bfff          4k 00000000 rw---Ip- (rwx) sed bss
0e15376ce000-0e15376cefff          4k 00000000 ------p- (rwx) guard
0e1547049000-0e154705efff         88k 00000000 r----Ip+ (rwx) ld.so.hints file mapping
0e154ad98000-0e154adcefff        220k 00000000 r----Ip+ (rwx) libc.so.97.0 rodata
0e154adcf000-0e154ae75fff        668k 00036000 --x-eIp+ (rwx) libc.so.97.0 text
0e154ae76000-0e154ae76fff          4k 000dc000 r----Ip- (rwx) libc.so.97.0 relro
0e154ae77000-0e154ae7cfff         24k 000dd000 r----Ip- (rwx) libc.so.97.0 relro
0e154ae7d000-0e154ae7efff          8k 000e2000 rw---Ip- (rwx) libc.so.97.0 data
0e154ae7f000-0e154ae7ffff          4k 000e4000 r----Ip- (rwx) libc.so.97.0 malloc page
0e154ae80000-0e154ae8dfff         56k 00000000 rw---Ip- (rwx) ...
0e154d973000-0e154d975fff         12k 00000000 rw---Ip- (rwx) ...
0e1570295000-0e1570295fff          4k 00000000 r----Is- (r--) ...
0e15bcd7d000-0e15bcd7dfff          4k 00000000 rw----p- (rwx) a memory allocation
0e158987f000-0e158987ffff          4k 00000000 ------p+ (rwx)  unmapped guard page
0e15d729c000-0e15d729cfff          4k 00000000 --x-eIp+ (rwx) sigtramp page
0e162f879000-0e162f87bfff         12k 00000000 r----Ip+ (rwx) ld.so rodata
0e162f87c000-0e162f87dfff          8k 00000000 -----Ip+ (rwx) ld.so boot.text destroyed
0e162f87e000-0e162f889fff         48k 00005000 --x-eIp+ (rwx) ld.so text
0e162f979000-0e162f979fff          4k 00011000 r----Ip- (rwx) ld.so relro
0e162f97a000-0e162f97afff          4k 00012000 rw---Ip- (rwx) ld.so data
0e162f97b000-0e162f97bfff          4k 00000000 rw---Ip- (rwx) ld.so bss
7f7ffdeee000-7f7fff7edfff     25600k 00000000 -----Ip+ (rwx) stack growth area
7f7fff7ee000-7f7ffffedfff      8192k 00000000 rw-S-Ip- (rwx) stack
```

Immutable mapping

- At least 2 attacks have manipulated mmap(2) or mprotect(2) to change a permission, perform a memory operation, and continued to control/escalation
- New system call mimmutable(2) allows locking the permissions of a region
  - No mprotect(2). No mmap(2) or munmap(2) which might replace the object
- Not normally called by programs themselves
- Kernel does this in execve(2) for a few regions
- ld.so takes care of main program and library mappings where suitable
- Only carefully chosen regions are made immutable

Immutable - Implementation

- 6 months of work
  - RELRO activation made me pull my hair
  - TEXTREL binaries required a similar workaround
  - malloc(3) self-protection interaction
  - Chrome v8 flags self-protection interaction

- Foundation for some other Synthetic Protections:
- It becomes possible to cache addresses, because the specific objects cannot be replaced!

X without R: Execute-Only Permission

- Newer processors have MMU or features which can enforce Execute-Only (we call it Xonly)
- We avoided working on this because only a few machines had MMU support, and it requires toolchain / application repair
- iOS is execute-only; Android tried a few years ago (abandoned)
- Time for OpenBSD to do it
- We found & fixed the missing steps, transitioned most platforms, and found a few MMU mechanisms along the way

Xonly: Fix userland

- Tools
  - Compilers – data islands, jump tables, etc
  - Linkers, correct placement seperation
- Applications
  - Dumb applications that invent their own ABI (very few)
  - Chrome, Node: V8 – the embedded blob
  - FFI
  - OpenSSL libcrypto, and so many copies..
- Concurrent development, 10 people, 12 weeks

Xonly: Machine-independent kernel support

- execve(2) ELF parser has to become strict
- Kernel does some Xonly enforcement, ld.so and crt0 do others
- Text-relocation binary support
- Some interaction with **Immutable** Permission

- Some uvm / pmap page permissions transitions were not anticipated and code needed repair

Xonly: X text without MMU support

- Many cpu families have members with & without MMU support
- A surprising synthetic behaviour!
- If cpu has independent R and X fault indicators, we can notice a R operation (which faults up to vm layer) which happens before a X operation (which could be in the MMU/TLB)…
- So some reads will be blocked

```
/usr/lib/libc.so.97.0 d12e8c2c-d13ad9a8 (c4d7c, 197 pg) prot X
        yynnnnnnyyyyyyyyyyyyyyyyyyynnnnyyyyyyynnnnnnnnnnnnnnnnnnnnnnnnnn
        nnnnnnnnnyyyyyyynnnnyyyyyynnyynnnynnyyyynnnnyyyyyyyyyyyyynnnnnnn
        nnnnnnnyyyyyynnnnyyyyyynnnnnnyyyyynnnyyyynnnnyyyyynnyyyyyyyyyyyy
        yyyyy
        read 104 pages of 197
        cannot read the whole
```

Xonly: Kernel copyin-xonly for code regions

- 2 types of non-execution reads
  - Userland reads userland memory
  - Kernel reads userland memory
- The 2nd one is

```
write(1, &main, 4);
```

- Inside the kernel, this turns into

```
copyin(useraddr, kern-buffer, size);
```

- Blocks reading code areas
- Blocks BROP (Hacking Blind)

child

	userland	kernel
ld.so	unreadable	unreadable
mmap xz	unreadable	unreadable
mmap x	unreadable	unreadable
mmap nrx	unreadable	unreadable
mmap nwx	unreadable	unreadable
mmap xnwx	unreadable	unreadable
main	readable	unreadable
libc unmapped?	unreadable	unreadable
libc mapped	readable	unreadable

parent

	userland	kernel
ld.so	readable	unreadable
mmap xz	unreadable	unreadable
mmap x	readable	readable
mmap nrx	readable	readable
mmap nwx	readable	readable
mmap xnwx	readable	readable
main	readable	unreadable
libc unmapped?	readable	unreadable
libc mapped	readable	unreadable

Xonly: Kernel copyin-xonly for code regions

- Per-process, Kernel maintains a 2-4 entry mini-cache of text (code) sections marked Xonly
  - Addr,len ranges can be cached because these regions have Immutable Permission
  - Main program text, sigtramp, ld.so text, libc.so text
- Mini-cache cannot be expanded by userland process
- libc.so range is learned when ld.so calls msyscall(2) for Syscall Permission
- Checked before every copyin(9), on machines without MMU support
- Checking cost is below the noise floor

Xonly: hardware support

- ARM64, RISCV64 have proper RWX bits
- HPPA has a strange gateway feature
- Sparc64 SUN4U has split I and D TLB, with software loading
- Newest MIPS (octeon) have a Read-Inhibit bit (Valid implies R or X, but RI disables R, much like x86 NX)

- The surprise: Newer Intel/AMD cpus can do Xonly

Xonly: amd64 PKU

- A fairly new CPU feature: cpuid to detect + register to enable
- PTEs contain new 4-bit PK value, indexing into RPKU register which contains 16 2-bit blocks (WI = write inhibit, RI = read inhibit)
- We leave regular memory as PK=0, with matching RPKU bits WI=0, RI=0
- Xonly pages are marked PK=1, with RPKU bits set to WI=1,RI=1
- So, kernel pre-loads RPKU value 0xfffffffc

Xonly: amd64 PKU

- But userland can change the RPKU register!
- On every kernel entry, if the RPKU register has been changed kill the process
- We get 99.9% effective Xonly

Xonly: other PKU

- PKU idea was inherited from IBM mainframes
- So powerpc G5 & powerpc64 also have a PKU feature
- On these processors userland can be blocked from changing the register

Stack Protection

- New Protection Mechanism:
  - When a process does a system call, the SP register MUST point to stack memory!
  - If it does not, we assume a ROP / ROP Pivot, and kill the process
- Kernel execve() sets up the stack + stack grow region, but mmap(2) gains a MAP_STACK flag
- pthread stacks are a bit tricky
- sigaltstack(2) is worse, new rule required: stacks must be new all-zero mapping, so that no underlying data persists

Execute Syscall Protection

- New Protection Mechanism:
  - When a process does a system call, the PC must point inside a region where system calls are permitted
  - If this is violated, process is killed

- 2 to 4 regions, 2 cases:
  - Static: main program text section, sigtramp page
  - Dynamic: ld.so text section, sigtramp page, and ld.so adds libc.so text using msyscall(2)
- Cannot create a PROT_EXEC region to perform system calls

Stack and Syscall Protection - Implementation

- Per-process, there are only a few valid regions
- For **Stack** and **Syscall**, kernel maintains a start, length, and serial
- Serial is incremented everytime a relevant mapping is changed
- If serial has changed, re-learn from vm system (more expensive operation)

- Expected a small performance impact
- Worst-case test programs saw tiny performance impact
- But real-world application impact was below the noise floor

Stack and Syscall Protection - Justification

- ROP attack code is really weird
- Bizzare execution restrictions result in bizzare actions
- **Stack** and **Syscall** Protection detect a variety of easier exploit patterns, pushing the ROP programmer to explore more challenging schemes, which may not be viable
- Increasing exploitation difficulty is a valid strategy

Procmap of sed(1)

- Sample output (ediedt)
- Removed most malloc(3)
- Notice:
  - Random layout
  - X without R
  - Many I (Immutable)
  - Some e (Syscall)
  - Unmapped guards
  - S (Stack) near end

```
Start            End                 Size Offset    rwxSeIpc  RWX   Object
0e1330a60000-0e1330a62fff           12k 00000000 r----Ip+ (rwx) sed rodata
0e1330a63000-0e1330a68fff           24k 00002000 --x--Ip+ (rwx) sed text
0e1330a69000-0e1330a69fff            4k 00000000 r----Ip- (rwx) sed relro
0e1330a6a000-0e1330a6afff            4k 00000000 rw---Ip- (rwx) sed data
0e1330a6b000-0e1330a6bfff            4k 00000000 rw---Ip- (rwx) sed bss
0e15376ce000-0e15376cefff            4k 00000000 ------p- (rwx) guard
0e1547049000-0e154705efff           88k 00000000 r----Ip+ (rwx) ld.so.hints file mapping
0e154ad98000-0e154adcefff          220k 00000000 r----Ip+ (rwx) libc.so.97.0 rodata
0e154adcf000-0e154ae75fff          668k 00036000 --x-eIp+ (rwx) libc.so.97.0 text
0e154ae76000-0e154ae76fff            4k 000dc000 r----Ip- (rwx) libc.so.97.0 relro
0e154ae77000-0e154ae7cfff           24k 000dd000 r----Ip- (rwx) libc.so.97.0 relro
0e154ae7d000-0e154ae7efff            8k 000e2000 rw---Ip- (rwx) libc.so.97.0 data
0e154ae7f000-0e154ae7ffff            4k 000e4000 r----Ip- (rwx) libc.so.97.0 malloc page
0e154ae80000-0e154ae8dfff           56k 00000000 rw---Ip- (rwx) ...
0e154d973000-0e154d975fff           12k 00000000 rw---Ip- (rwx) ...
0e1570295000-0e1570295fff            4k 00000000 r----Is- (r--) ...
0e15bcd7d000-0e15bcd7dfff            4k 00000000 rw----p- (rwx) a memory allocation
0e158987f000-0e158987ffff            4k 00000000 ------p+ (rwx)  unmapped guard page
0e15d729c000-0e15d729cfff            4k 00000000 --x-eIp+ (rwx) sigtramp page
0e162f879000-0e162f87bfff           12k 00000000 r----Ip+ (rwx) ld.so rodata
0e162f87c000-0e162f87dfff            8k 00000000 -----Ip+ (rwx) ld.so boot.text destroyed
0e162f87e000-0e162f889fff           48k 00005000 --x-eIp+ (rwx) ld.so text
0e162f979000-0e162f979fff            4k 00011000 r----Ip- (rwx) ld.so relro
0e162f97a000-0e162f97afff            4k 00012000 rw---Ip- (rwx) ld.so data
0e162f97b000-0e162f97bfff            4k 00000000 rw---Ip- (rwx) ld.so bss
7f7ffdeee000-7f7fff7edfff        25600k 00000000 -----Ip+ (rwx) stack growth area
7f7fff7ee000-7f7ffffedfff         8192k 00000000 rw-S-Ip- (rwx) stack
```

One more: pinsyscall(SYS_execve)

- This new Permission is smaller than a page
- `pinsyscall(SYS_execve, &execve, libcstublen)` is called at program startup [in either ld.so or crt0]
- Then execve(2) may only be called from inside the specific system libc call stub (which is generally less than 80 bytes long)

- Before this, ROP attackers could use any syscall instrution they find [in main program, ld.so, sigtramp, libc, or polymorphic on variable-size instruction architecture] to reach execve(2)
- Address caching depends upon Immutable Permission

ROP attacker’s situation now

- Stack damage → want to ROP → and then problems:
  - Cannot find as many (or any) gadgets: ASLR, random relink, reduction, RETGUARD removed tail gadgets
  - Cannot perform system call from SP or PC pivoted positions
  - Cannot mutate memory permissions
  - Cannot scan address space for some types of info leak
  - Cannot reuse a known syscall location in ld.so to reach execve
  - …
  - Immutable mappings may help with other inexpensive checks

All mitigations on one page

W^X stack-protector (stack damage detect) .rodata-use

ASLR library-random-relinking library-random-order-mapping

fork+exec policy

SROP-blocking setjmp-cookie

RETGUARD (tail CFI, stack overflow detect, 100% coverage)

x86 polymorphic gadget reductions

syscall PC & SP checks, execve stub check

mimmutable, xonly, xonly emulation

Conclusion & Questions

We should push attackers towards methods

- requiring more intense labour
- requiring features which are disrupted
- with worse success rates

All these Mitigations try to achieve these goals

Real World impact will be judged in coming years

„My attack didn't work on OpenBSD but it worked on Linux"

Hacker77, September 2031