Exploit Mitigation Techniques

An update after 10 years

Theo de Raadt
The OpenBSD Project

In 2000, OpenBSD started a development initiative to intentionally make the process memory environment LESS PREDICTABLE and LESS ROBUST without impacting well-behaved programs.

It has taken more than 10 years to finish the task.

Most of these ideas are now adopted by other systems.

A status report is needed.

Agenda

The problem

The concept

The methods

Enabled by default

Adoption by others

Successes and regrets

It is always the same story:

The clever attacker

... finds a bug which damages memory (overflow, etc)

... analyses the unintended side-effects created

(and because of the strict regularity of the system environment)

... easily crafts an exploit which grants him advantage.

Concept: How can we increase resiliance?

The process memory environment is made up of a mix of well-defined and undefined behaviours.

How much of the 'undefined' can we change and have things still work?

Don't want to break normal/expected behaviours

But maybe change anything else which makes exploits hard/impossible?

As long as the performance cost is insignificant / very low..

Methods

Summary of methods

- Provide an unpredictable resource base with minimum permissions
  - random stack gap
  - program segment mapping randomization
    - shared library ASLR, random ordering
    - PIE
    - mmap ASLR
  - increase use of .rodata
  - malloc randomizations

- Where it is possible to spot damage, fail hard
  - stack protector
  - stackghost
  - atexit / ctor protection
  - etc.

Methods: a significant caveat

Any one mechanism (alone) may be insufficient to stop an attack:

- People have found ways around ASLR (in isolation)
- People have found ways around W^X (in isolation)
- SSP does not discover all types of frame damage
- Too much address-space randomizaton -> fragmentation in kernel page management -> performance lost, feature gets disabled...

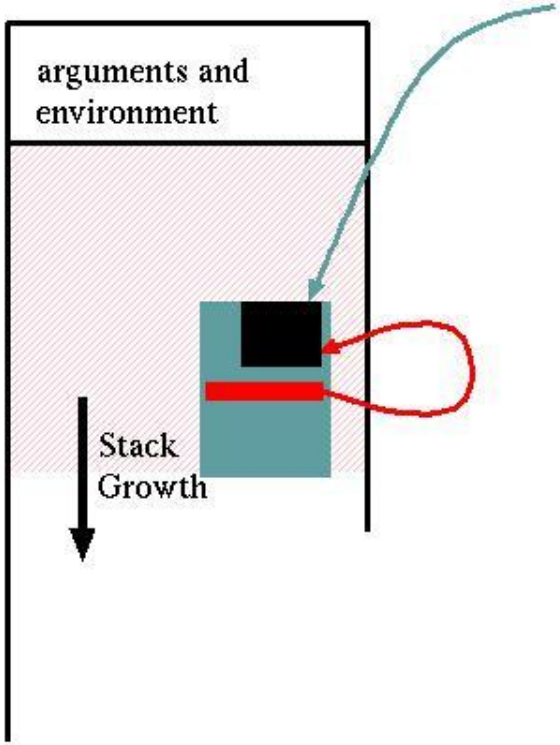
No mathematical proof that a collection of these mechanisms blocks attacks

But that is not same as saying "Don't try"

With those thoughts in mind, we started work and deploying them in OpenBSD releases.

Most common attacks rely on damage of the local stack frame

The mechanics of a (simple) stack-based buffer overflow



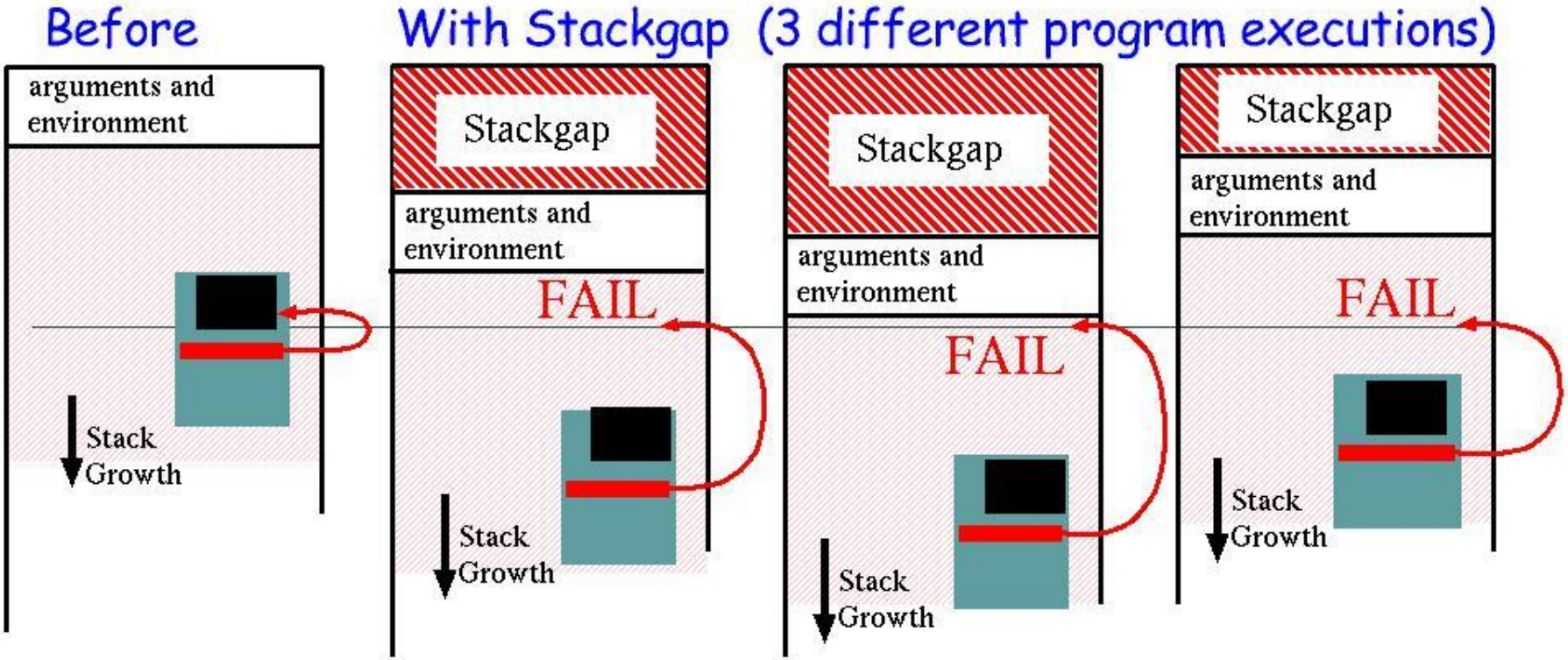
Attacker overflows buffer on stack
Note: Buffer is ALWAYS at the same place

Overflow overwrites function return address -- fixed value pointer into overflow buffer - execution starts

Key point: The pointer is an absolute address.

Solution: a random-sized gap at top of stack (8-byte aligned)

Random Stack Gap



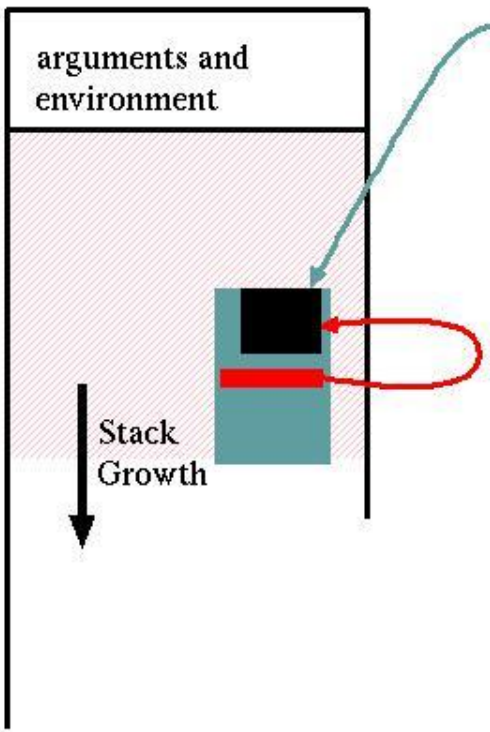
Wastes at most 1 page of real memory (the gap is virtual)

With a stackgap of 256K, attack feasibility is reduced to
1 in 2^(n - paddingrule)

This is a 3-line change to the kernel.

Introducing W^X: A better page permission policy

Many bugs are exploitable because the address space has memory that is both writeable and executable (permissions = W | X)



this location has to be executable for the exploit to work

We could make the stack non-executable...

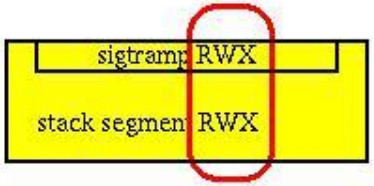
Hmmmm... how about a generic policy for the whole address space:

A page may be either writeable or executable, but not both (unless the program specifically requests)

We call this policy W ^ X (W xor X)

Let's see how far we can apply it!

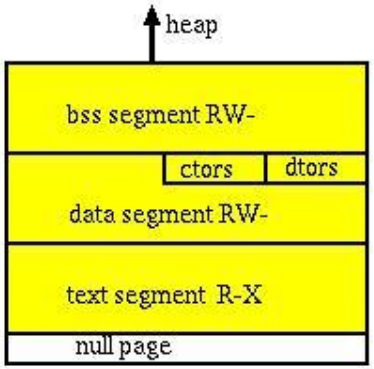
W^X transition: Introduction to static binaries



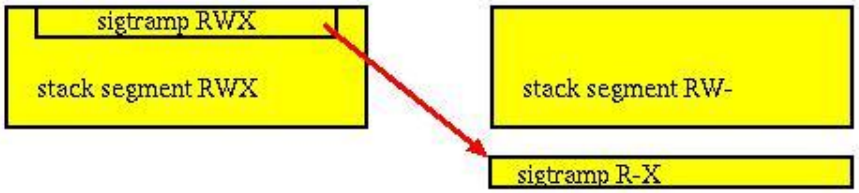
This is what static executables used to look like in memory.

The stack has a piece of executable called the "signal trampoline"

First problem: The stack is executable



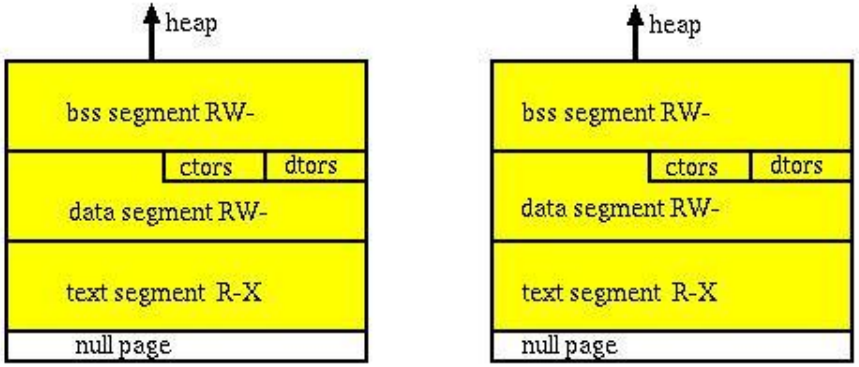
W^X transition: Sigtramp separation



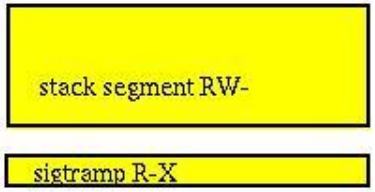
First we move the signal trampoline away from being the top page of the stack (to a per-process random address)

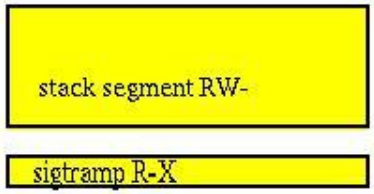
The stack becomes non-executable

But perhaps we should look at what shared libraries do, next..

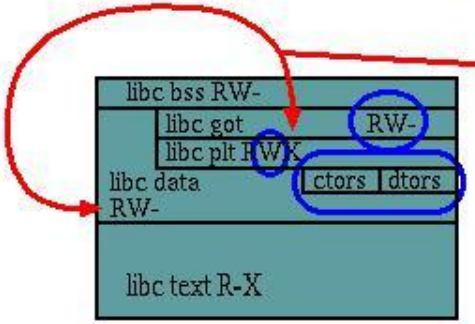


W^X Transition: Intro to dynamic binaries



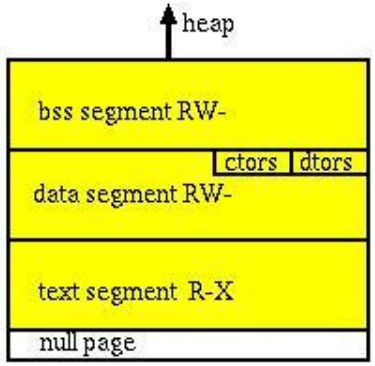


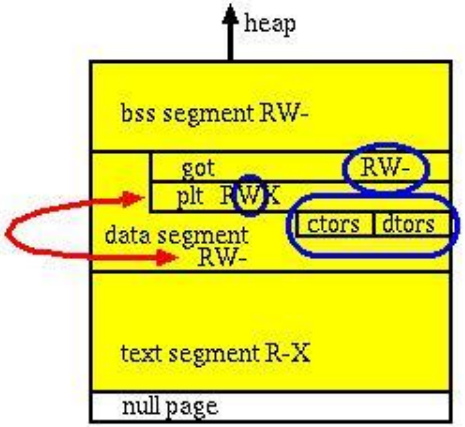
An example of how shared libraries (used to) map



Note the "data" segments which are supposed to be only RW- but contain objects which are RWX

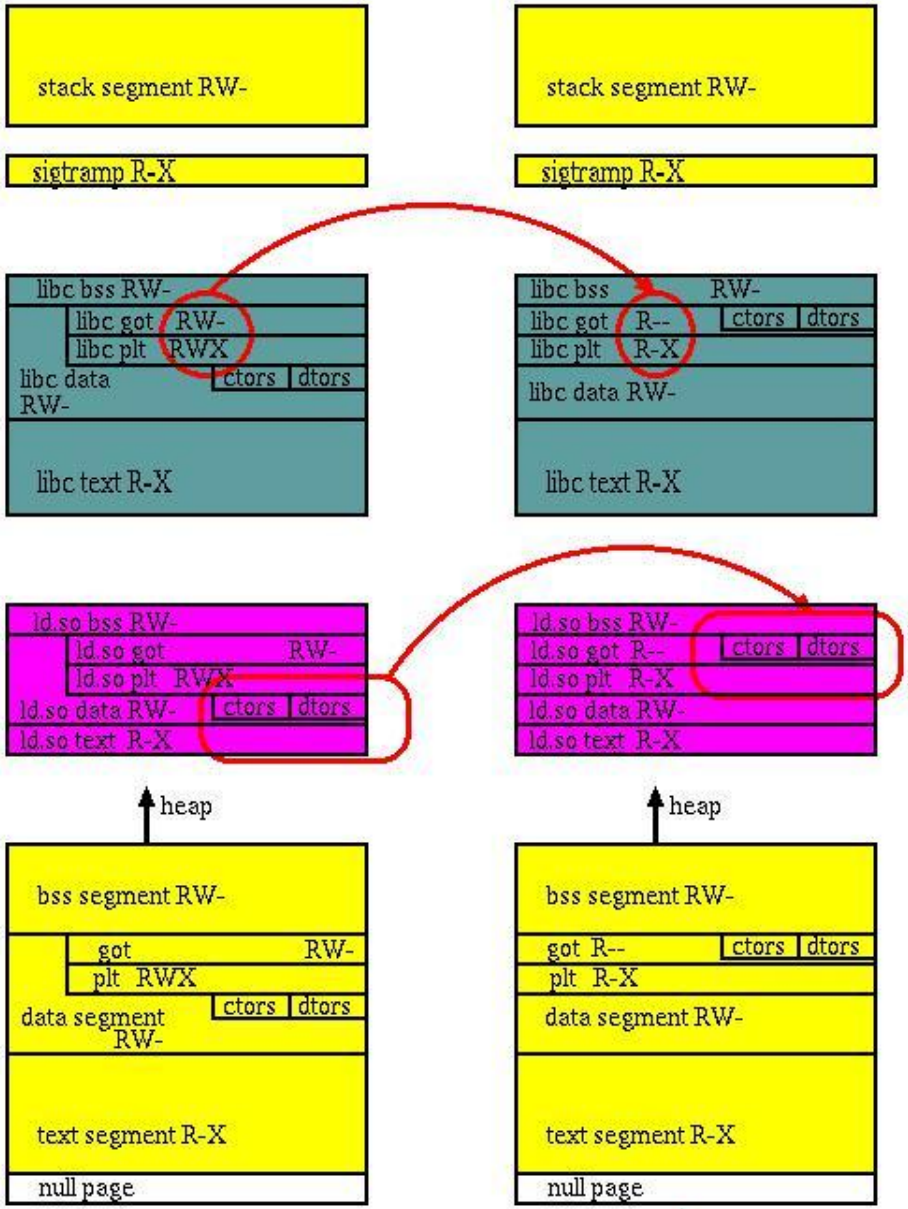
An additional danger is that some objects are writeable when they do not need to be, ie. GOT/PLT/ctors/dtors





GOT = shared lib Global Offset Table
PLT = shared lib Procedure Linkage Table
ctors = c++ constructors
dtors = c++ destructors

W^X Transition: Applying policy to GOT/ctor/dtors



GOT and PLT get their own pages and become non-writeable (and we teach ld.so how to cope)

dtors/ctors move in with the GOT, thus become non-writeable

Now the data segment has no objects with X permission!

We made a few things non-writeable (for free)

No page has both W and X bits! Policy achieved.

W^X Transition: The .rodata segment

Readonly strings and pointers were stored in the .text segment: X | R

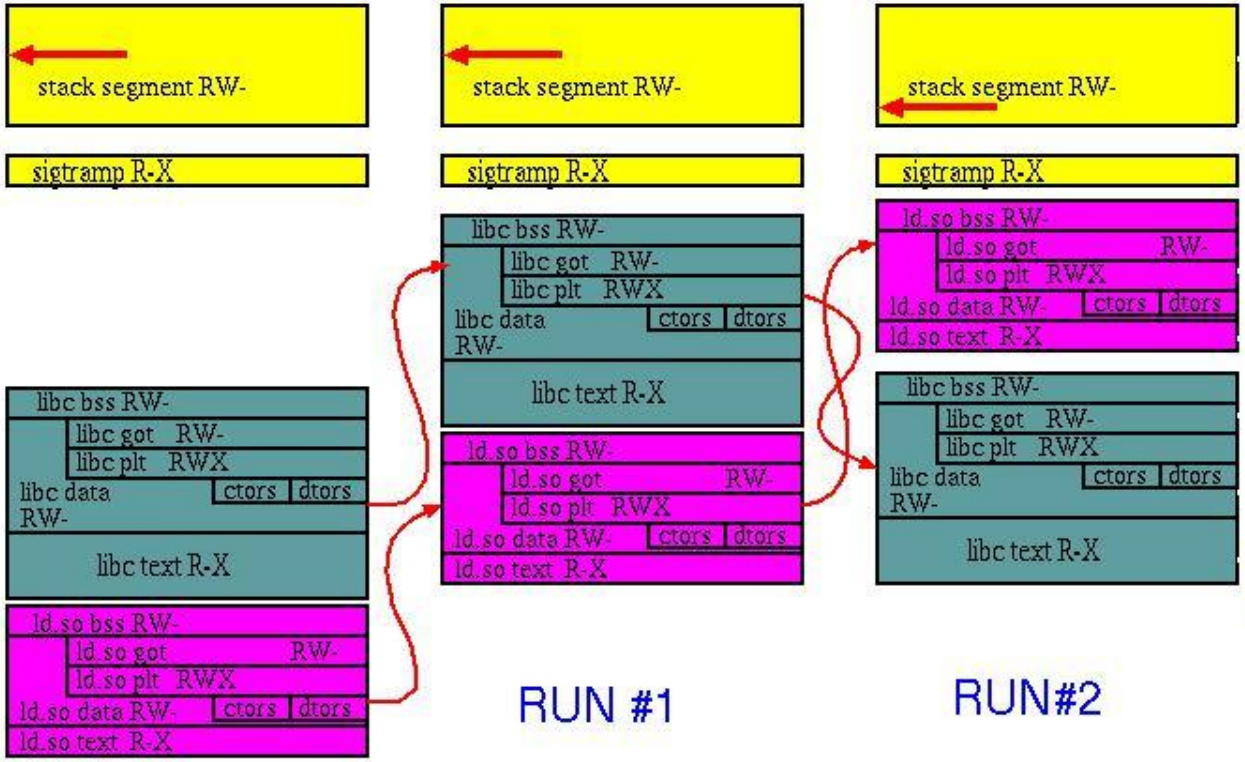
Meaning const data could be executed (could be code an attacker could use as ROP payload)

Solution: start using the ELF .rodata segment

These objects are now only R, lost their X permission

Greater policy: "minimal set of permissions"

ASLR: randomly map & order libraries

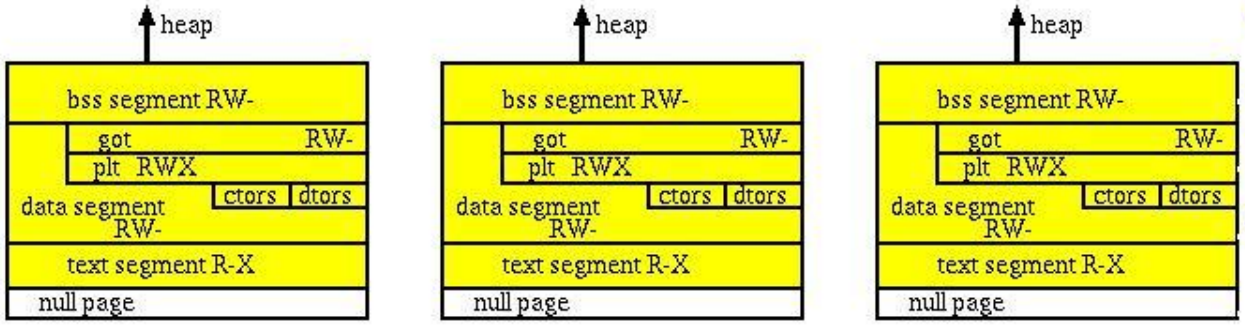


Perturb shared library mappings.

Base address ..

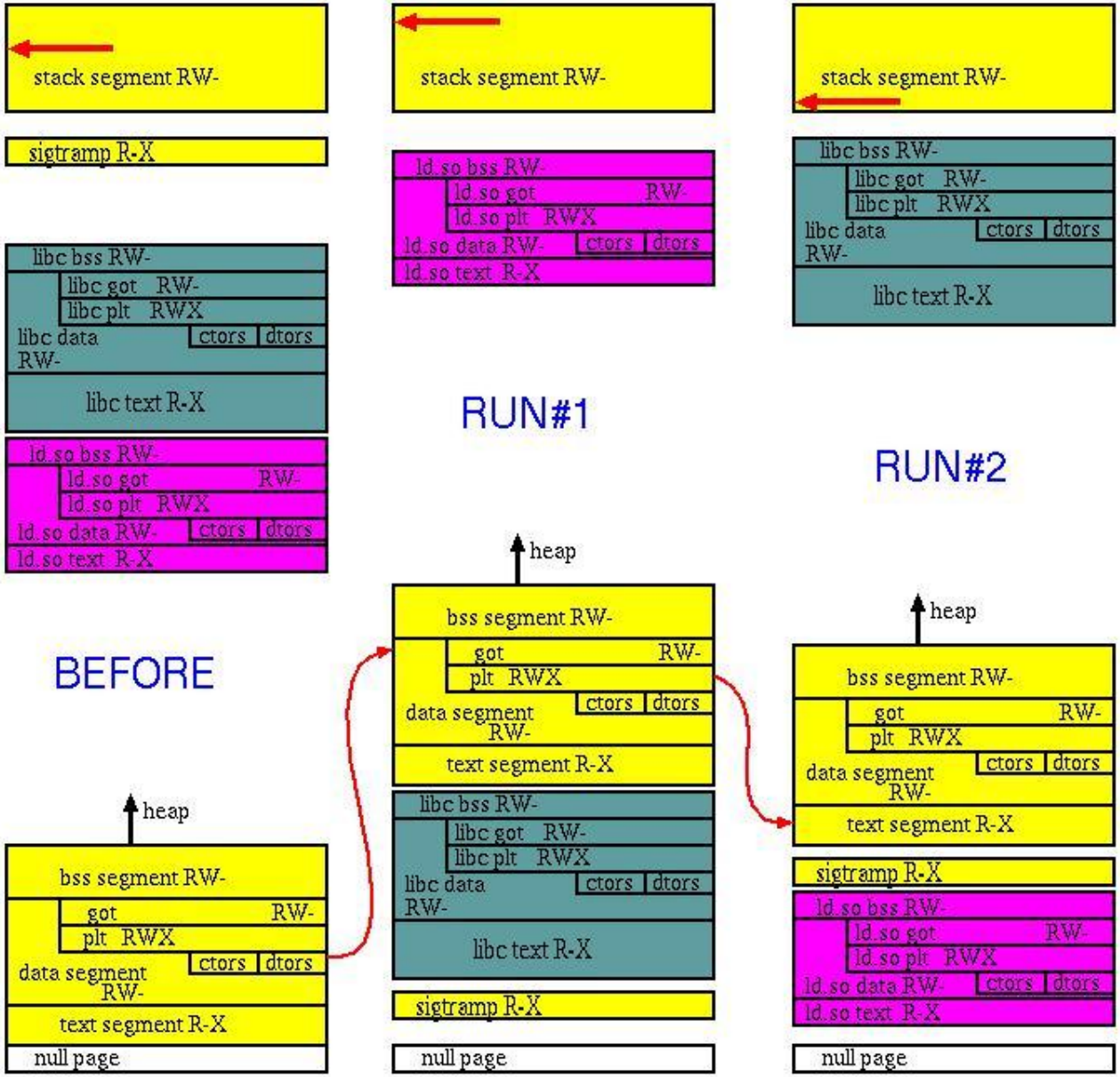
.. and order of mapping.

BEFORE



On each run, each library has a new address

PIE - Position Independent Executable



A compiler change called PIE makes the main program a "shared library"

Then we can map it anywhere

On each run, the main program has a new address

ASLR: Randomized mmap()

Each time you do an allocation using mmap()...

If MAP_FIXED is not specified, choose a random address

Result:

Each time you run a program....

... different address space behaviour!

ASLR: Randomized malloc()

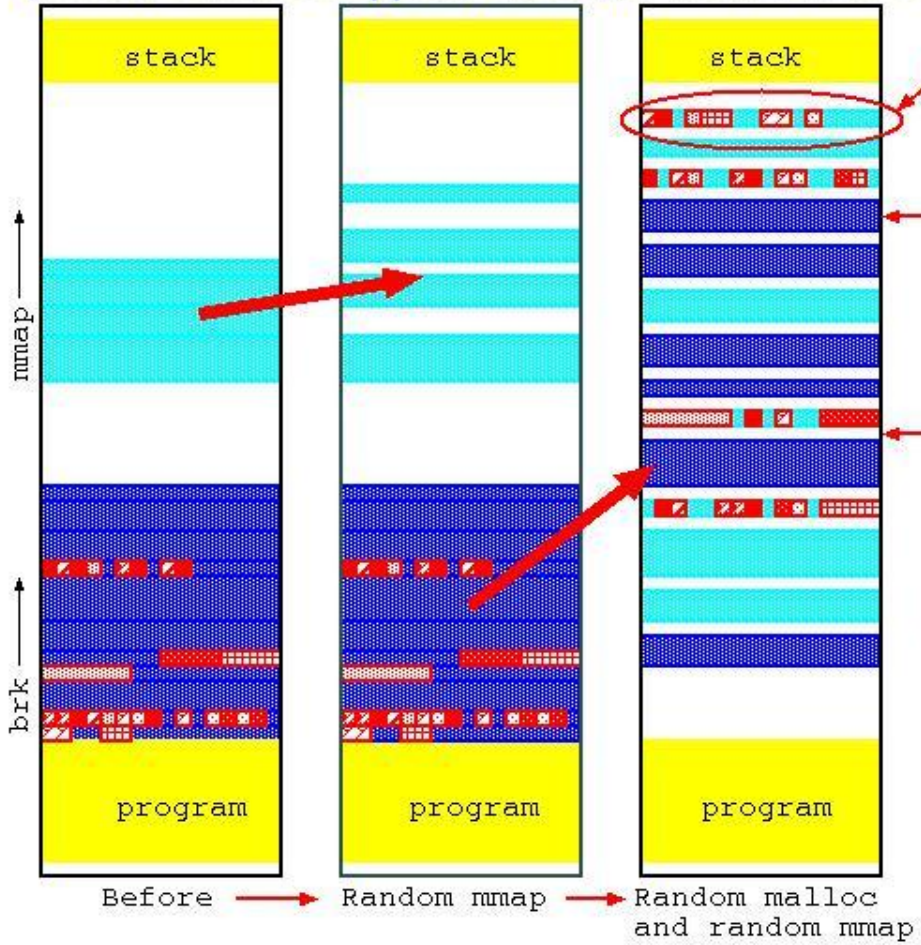
Did you know that the addresses of objects allocated by malloc() are fairly predictable?

Two types of objects are managed by malloc()

- Smaller than a page:
  - malloc() maintains buckets of "chunks"
  - Randomize chunk selection out of bucket
  - Enabled using /etc/malloc.conf 'G' option (still required?)
- Equal or greater than a page:
  - Rely on random mmap()

ASLR: mmap / malloc demo

(lots of details skipped, ie. PIE, shared libraries, etc)



tiny allocations done with malloc() are randomly allocated within a "bucket" page

allocations with malloc() over pagesize are simply allocated using mmap()

mmap() gives us "gap pages" (more on that later..)

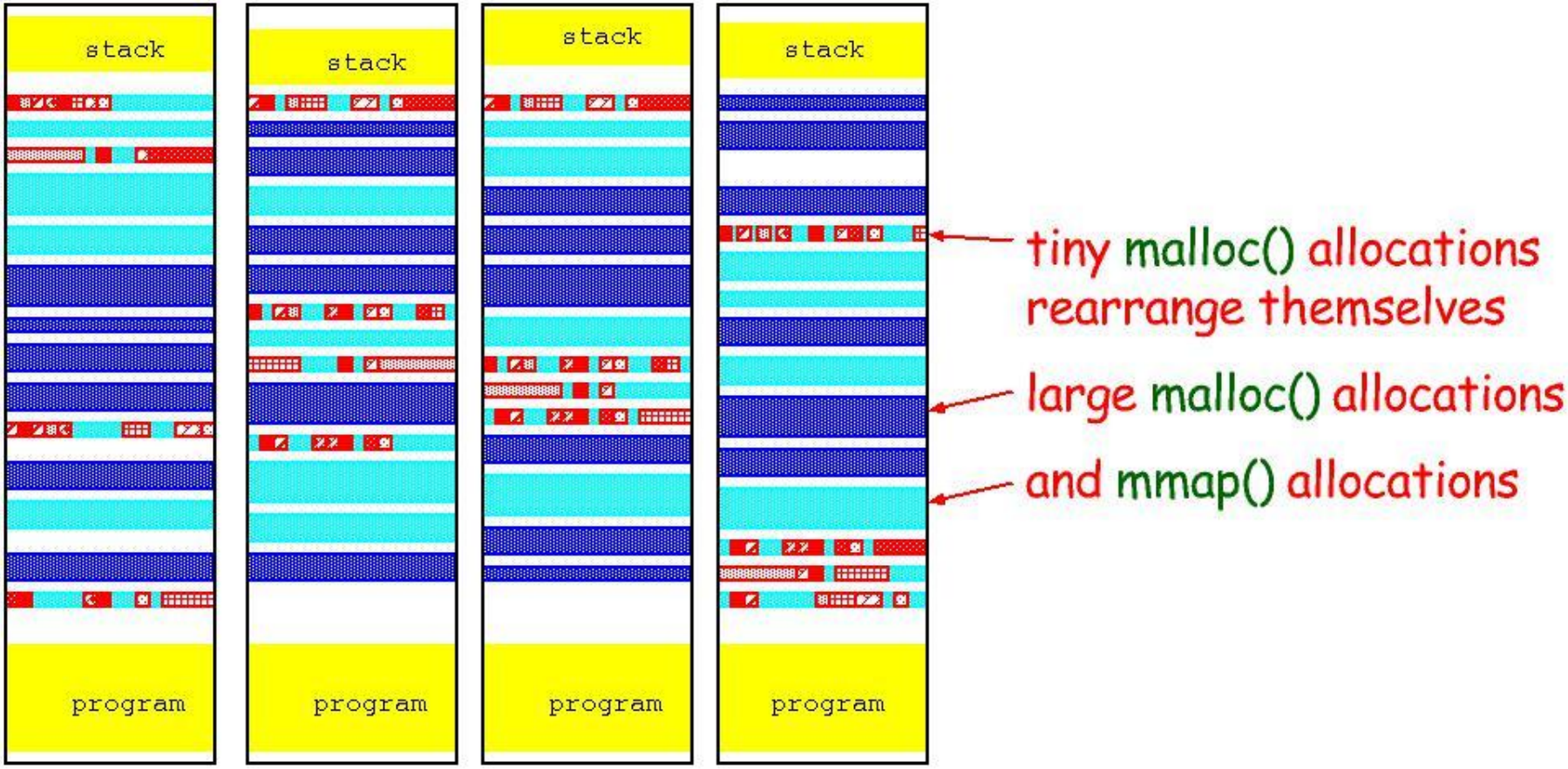
mmap()-allocated memory

malloc()-allocated memory (page size or greater)

malloc()-allocated memory (objects < page in size)

Randomized allocations..

Of course, each time you run the program the allocations change.



Note: Not showing the effects of many other changes, like shared library randomization, etc, etc

Other benefits of mmap() malloc()

When you free() an object >= pagesize

- it gets unmapped using munmap()
- Therefore, access after free() becomes a detectable crash

Detecting buffer overflow, over"read"

- If you try to read/write beyond the end of an object, maybe there is a guard page there?

Future goals

- Be more proactive with placing allocations next to guards
- Or using "padding" guards

malloc() ... UNFORTUNATELY

Unfortunately much software is written to very low standards

The more of these features we enable, the more bugs we run into

malloc.conf 'G' option:

"Guard". Enable guard pages and chunk randomization. Each page size or larger allocation is followed by a guard page that will cause a segmentation fault upon any access. Smaller than page size chunks are returned in a random order.

A few malloc features cannot be enabled by default yet

Stack Protector

Compiler modification which catches most common stack-smashing problems

Original: http://www.trl.ibm.com/projects/security/ssp/

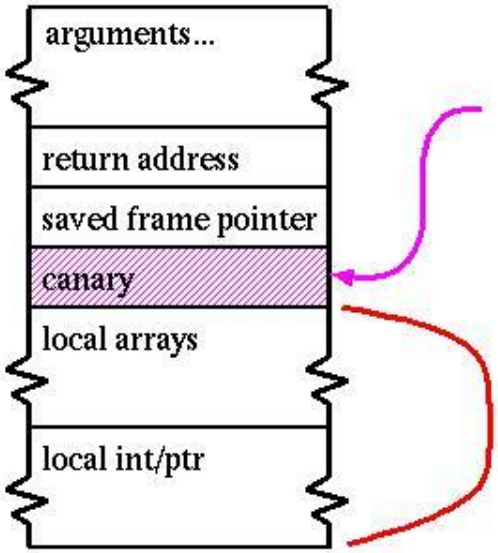
Compiler instruments generated code for each function:

- Prologue stores a random value (canary) on the stack
- Function Epilogue aborts if value has changed

Integrated into OpenBSD in December 2002

Industry (mostly Google) now making further enhancements

Stack Protector



A typical stack frame...

Random value is inserted here by function prologue ...

... and checked by function epilogue

Reordering: Arrays (strings) placed closer to random value -- integers and pointers placed further away

-fstack-protector-all compiled system is 1.3% slower at make build

BENEFITS SECURITY: Finds bugs and makes them unexploitable

VERY LOW COST: Every vendor should use it

StackGhost

A sparc/sparc64-specific buffer overflow protection mechanism

stackghost.cerias.purdue.edu/stackghost.pdf

These register-window architectures have backup-storage for the registers (reserved in each stack frame)

StackGhost is a scheme where the register-window SPILL/FILL trap handlers XOR the frame-pointer register (%i7) with a per-process random cookie ("wcookie")

Protects registers from modification while they reside in the stack frame (%i7 itself, or the next frame it points to...)

Like a weak Stack Protector... but zero cycle overhead.

W^X and !W: atexit() in libc

Introducing a case of minimum page permission use in libc

```
void exit(int status)
{
        struct atexit *p;          int n;

        for (p = __atexit; p; p = p->next)
                for (n = p->ind; --n >= 0;)
                        (*p->fns[n])();
        if (__cleanup)
                (*__cleanup)();
        _exit(status);
}
```

__atexit and __cleanup contain writeable function pointers!

We modified atexit(3) to maintain function pointer storage which is kept non-writeable

Confused yet?

Are you confused yet by all the layers of obfuscation?

The attacker is!

These changes combine to make exploitation very difficult.

Enabled by default

Strategy: as soon as something works... enable it and get everyone to use it!

Then: show upstream software projects the bugs we expose, and help verify the repairs they make

Only had to downgrade the aggressiveness once or twice:

A few revisions of ASLR (too greedy with address space)
malloc guarding (world is not ready for this)

Currently these methods do not restrict us from running any upstream software

Adoption: Most vendors

Microsoft has all significant mitigations fully integrated and enabled!!

Linux has code for all the mitigations. Most vendors enable them very sparingly (sshd), and in general support is disabled... :-(

Apple has ASLR (but not the other methods?)

Most Cell-phone platforms use these features, but less protection benefit (thread-intensive environments)

The upstream software ecosystem (ports) is ready and willing.

Adoption: FreeBSD

from wiki.freebsd.org/201309DevSummit

- Included in 10, but off by default:
  - stackgap randomization adds a random amount of empty space at the top of the stack
  - mmap randomization inserts a random gap between consecutive mappings
- Stack protection can now be enabled by default (but hasn't yet) after libc changes
- ldbase randomization discussed, but not implemented

Adoption: FreeBSD

from wiki.freebsd.org/201309DevSummit

- Included in 10, but off by default:
  - stackgap randomization adds a random amount of empty space at the top of the stack
  - mmap randomization inserts a random gap between consecutive mappings
- Stack protection can now be enabled by default (but hasn't yet) after libc changes
- ldbase randomization discussed, but not implemented

Adoption: FreeBSD

from wiki.freebsd.org/201309DevSummit

- Included in 10, but off by default:
  - stackgap randomization adds a random amount of empty space at the top of the stack
  - mmap randomization inserts a random gap between consecutive mappings
- Stack protection can now be enabled by default (but hasn't yet) after libc changes
- ldbase randomization discussed, but not implemented

Adoption: FreeBSD

from wiki.freebsd.org/201309DevSummit

- Included in 10, but off by default:
  - stackgap randomization adds a random amount of empty space at the top of the stack
  - mmap randomization inserts a random gap between consecutive mappings
- Stack protection can now be enabled by default (but hasn't yet) after libc changes
- ldbase randomization discussed, but not implemented

How can people go around saying FreeBSD is secure?

Summary

Low or non-existant performance hit -- all programs continue working

Exploitable problems have been found and fixed by our changes

Thousands of non-exploitable bugs were fixed also

These changes really stop attacks.

10 years: Wish more systems had adopted more of the features