Mitigations and other real security features

Largely invisible security improvements

Software will never be perfect

Erroneous condition logic fails, then cascades through successive failures, often externally controllable

Attackers can perform illegal access+control of code in program & libraries, or toy with kernel surface

Attacker‘s **knowledge** and **methods** are improving fast

Developer tools & practices insufficient at preventing bugs

We need multiple layers of defense, including **mitigations**

I work on mitigations

Mitigations are inexpensive tweaks which increase the difficulty of performing attack methods:

- low impact on normal operation
- huge impact during attack-scenario operation

Also, they act as pressure towards **robustness** in software.

When defect is detected, goal is to **Fail-Closed**

Robust (adj.)

”When used to describe software or computer systems, *robust* can describe one or more of several qualities:

- a system that does not break down easily or is not wholly affected by a single application failure
- a system that either recovers quickly from or holds up well under exceptional circumstances
- a system that is not wholly affected by a bug in one aspect of it“

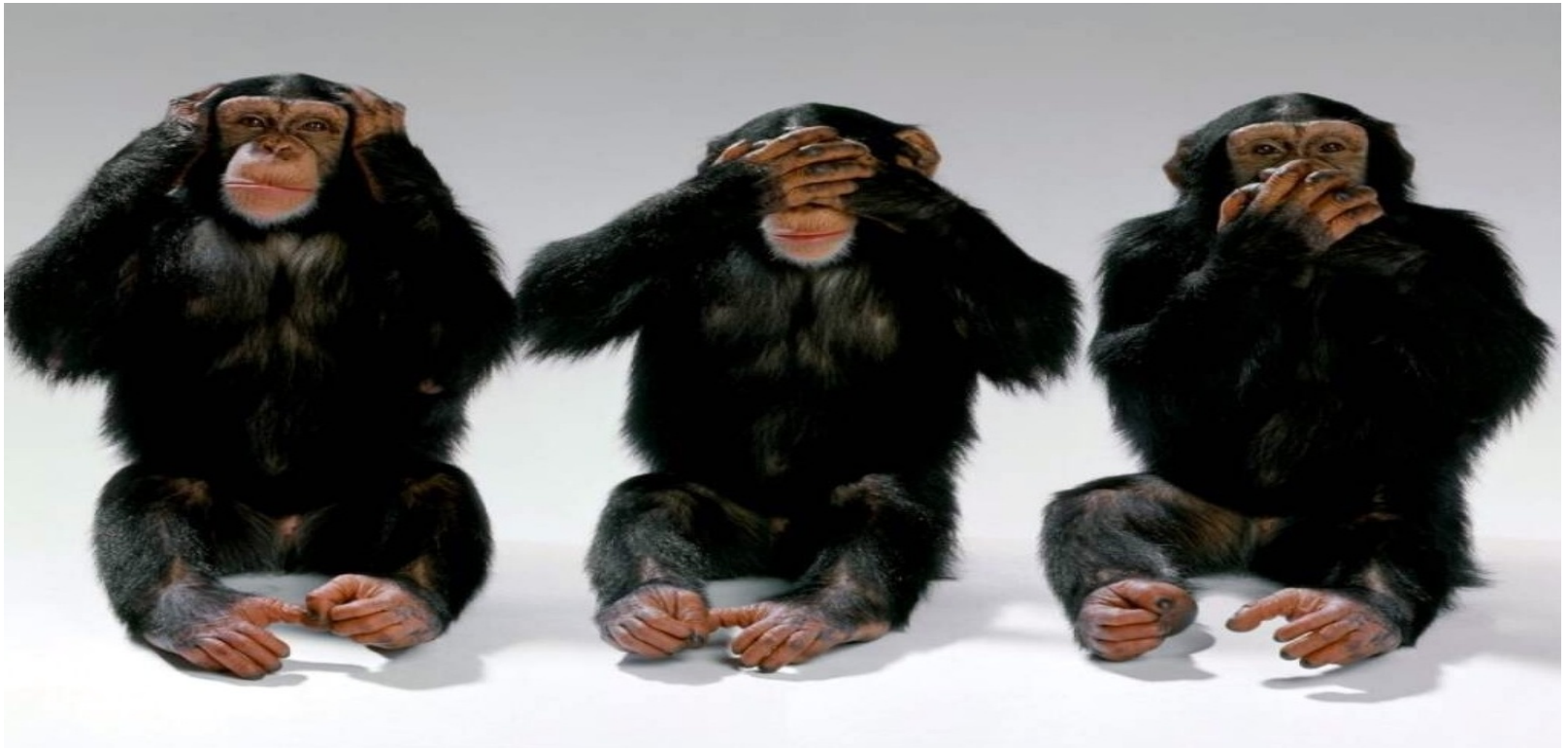
- **I strongly disagree – error-recovery is an unsound practice, and detection mechanisms do not exist**
- **If memory has been overwritten, how does one continue?**

Justifying Fail-Closed

1. User becomes aware of bug (in a painful way... but probably less painful than if an attacker does it first)
2. Hopefully some mitigation "detects-and-terminates" execution before control-flow creates a further mess of the stack, which may result in a weaker bug report
3. User files bug report
4. Programmer fixes bug, ships new software

This is Best Current Practice for building good software

Other approaches for handling bugs



Fixing bugs is what happens after you become aware

17 years of mitigation work

%rbx move
syscall sp check
Rev memcpy() detect
.openbsd.randomdata
Library-relinking
stackghost
poly-ret scrubbing
sendsyslog()
atexit()-hardening
otto-malloc()
Lots of arc4random
X-only .text?
W^X
kbind(2)
KARL
PIE
random KERNBASE
ASLR
pledgepath()
Kernel W^X
sigreturn() SROP cookie
RELRO
privsep
setjmp() cookies
X-only kernel?
pledge()
StackProtector
RETGUARD..
privdrop
per-DSO StackProtector
...RETGUARD4
sshd relinking?
cc deadreg-clearing
trapsleds
guard pages
fork+exec (never reuse an address space)

These changes cause "weird" or un-standardized operations to fail-closed (crash right now, please)

Features of mitigations

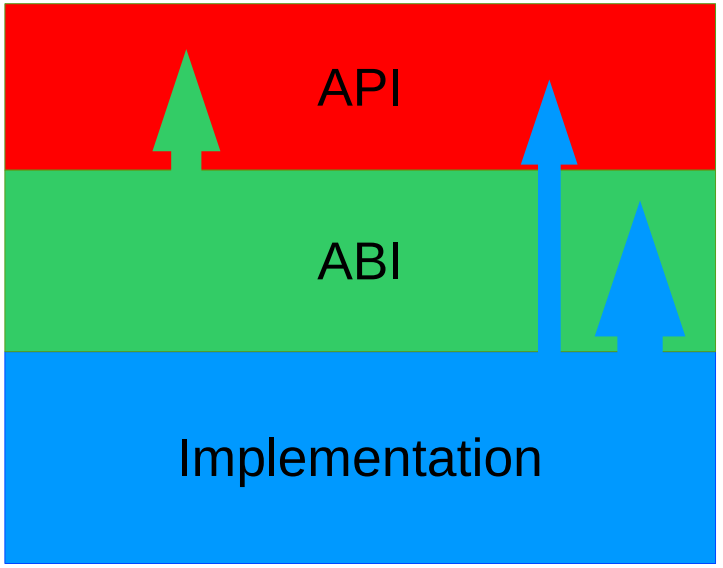
- Diminishes effectiveness of a specific attack method
- Efficient, low overhead
- Easy to understand
- Easy to incorporate/apply to old & new code
- One mitigation doesn‘t need to fix ALL the problems

- Exposes bugs earlier – improves triage

Components attackers use

Knowledge	+	Mechanism	+	Act upon Objects
A software bug or two..		ROP, etc		Filesystem
Substantial consistancy		Code Reuse		open fd‘s
Location of objects (relative and absolute)		Syscalls		Leftovers on the stack
Gadgets, constants, pointers, regvalues, etc.				

Knowledge: API, ABI, and beneath



Details of ABI and Implementation sneak upwards, providing attackers with data to use during attack

These details land on the stack and in registers, and get left behind

Left behind long enough for attackers to use...

ABI+Implementation leakage

```
{
    int i;
    void *p = &i;
```

If a crash happens in following code, you can find the return address based upon the register **p** is stored in.

Such low-level details are well-known, defacto-standardized, and exposed for performance reasons – but need it be so always?

Knowledge: stack, register leaks

Implementation details dribble onto the stack and registers

Attackers run the same binaries as you, in a nearly identical environment – therefore they can easily determine what-is-where – this is the surface ROP operates upon

When a bug crashes, they know the regs & stack offsets where many deterministic (integer and pointer) and relative-deterministic (pointer offset) leftovers are found

They just need mechanisms to pivot into control

Attacker tooling: ROP

Hijack instruction sequences using false return frames

Gadget is any small sequence of register/memory transfers above a true ret (or polymorphic ret) instruction

Attacker needs to know where gadgets are, and address of the stack

Other methods: JOP, SROP, etc.

Even scarier: BROP – Blind ROP

An address-space oracle

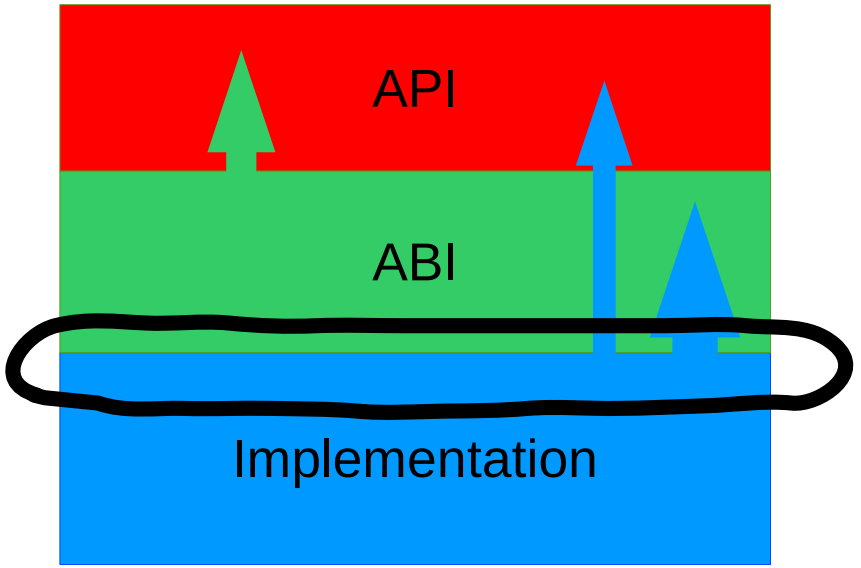
Repeated probes against reused address-space learns enough to perform minimum ROP operations

Can discover (and thus bypass) certain mitigations

Then, continue using various ROP methods...

(BROP is defeated by never-reuse-an-address-space)

Stack protector example

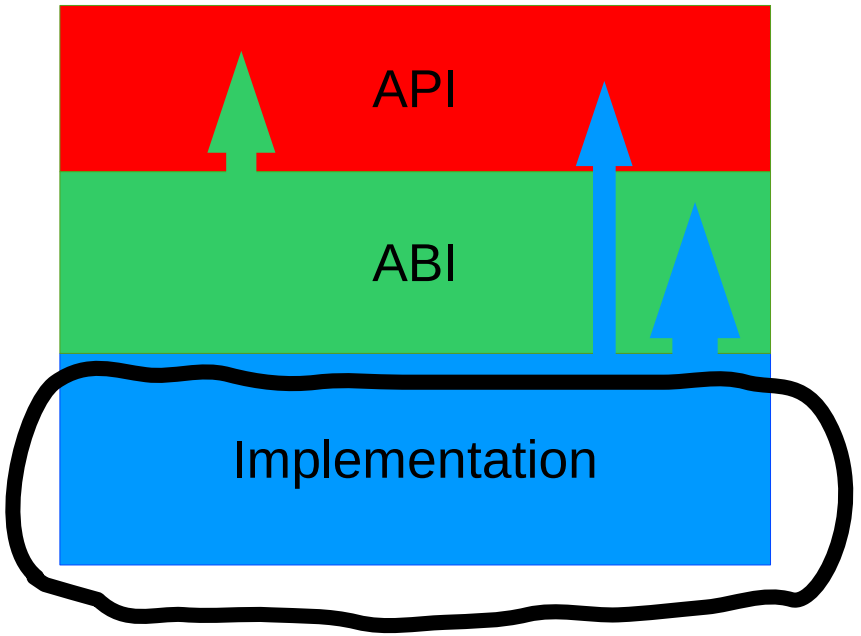


Attacker depends on knowing:
- Location in memory of stack
- ordering or stack objects
- callframe ordering:
  (in-args, savepc, savefp, locals)

Stack protector disturbs the simplest attack methodologies

(We have a better strack protector brewing)

ASLR effect?



Attack methods use 3 kinds of values found in stack/registers:
- absolute integer values
- relative addresses (offsets)
- absolute addresses (pointers)

To 3 places: code, data, stack.

ASLR increases difficulty of attack

Costs – time for a discussion

- Not all migitations will be performance-cost-free
- Processors have gotten faster
- What % of performance would you spend on security?
- Most mitigations are a reimplementation of an existing mechanism, just more strictly

Shared library improvements

Progressive improvements – in the old days the GOT & PLT were writeable!

- Secure-PLT
- No more relocs
- RELRO
- kbind()
- DT_BIND_NOW

Takes decades to add strictness / security

Malloc hardening

OpenBSD malloc hardened in 20 ways or more

- significant randomness, object rotation, meta-data protection, double-free and use-after-free detection, heap-overflow detection, …

Half of hardening features enabled by default

(Expensive ones not enabled, but available during devcycle)

Also API hardening: reallocarray(), recallocarray() freezero()

Pledge: Realistic POSIX subsets

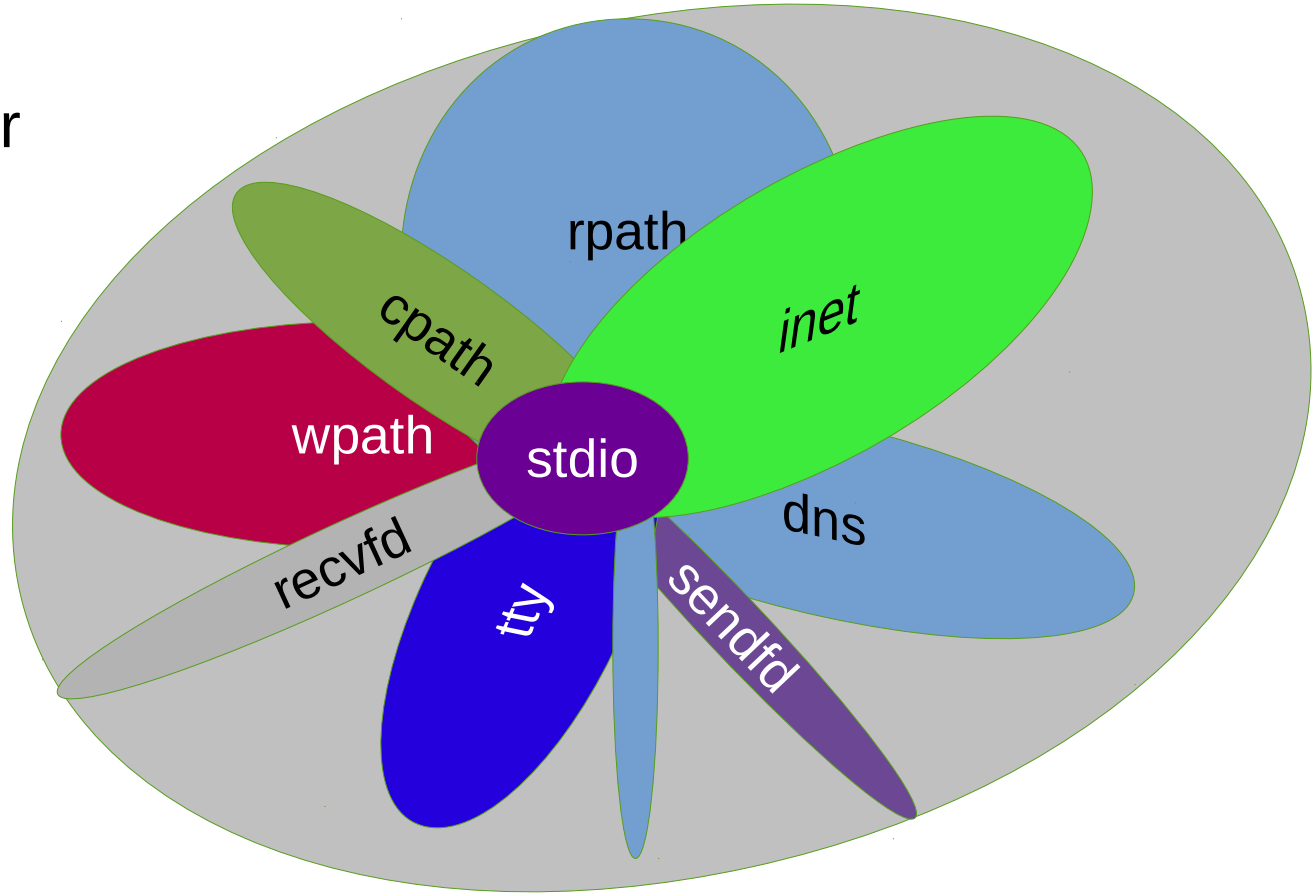
No subtle behaviour changes

No error returns

Fails-closed

Illegal operations crash

Easy to learn



How does pledge help?

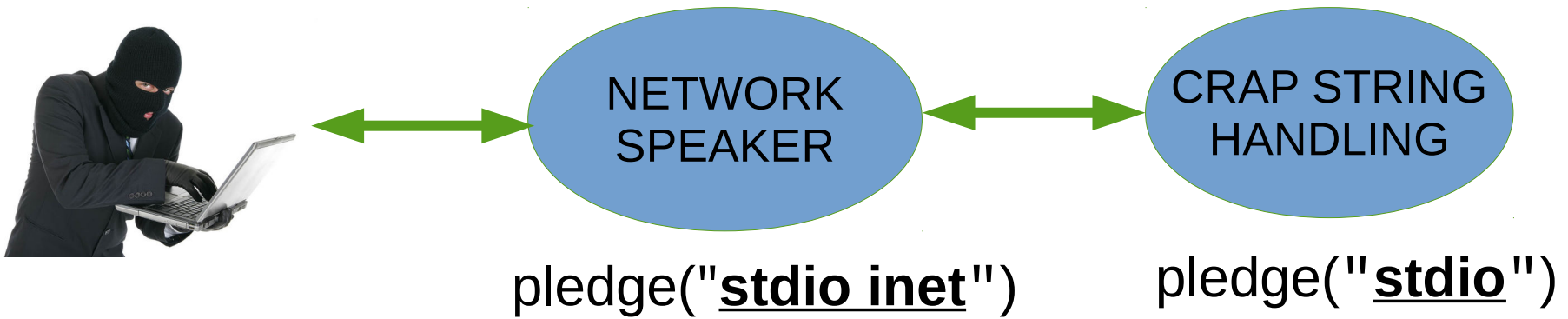
2nd specification of a program‘s behaviour and requirements is embedded directly into the program.

No behaviour changes -- only detects that the programmer‘s 2nd specification has been violated

2nd specification allows programmer to test their assumptions about system call use

Privilege Separation + Pledge

Pledge does an excellent job ENFORCING the security-specialization when a task is privilege-seperated:



Common ideas behind mitigations

- Reduce externally-discoverable knowledge
- Improve historical weaknesses of permission models
- Disrupt non-standard control-flow methods
- Education of seperation-of-duty during program design
- Reduce syscall availability during execution

Mitigations change details which are not specified by any standard

In ways which seem to harm current attack methods, and help us debug code

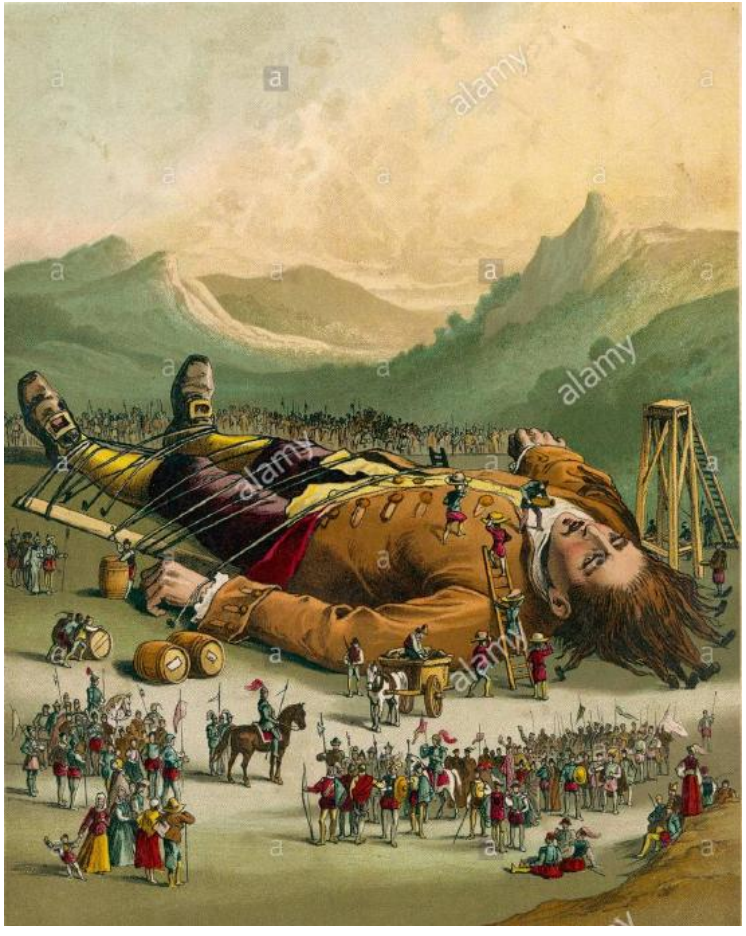
Resistance against mitigations

- Some await language/compiler advances to save us all
- Some prefer long-term binary-compatibity over attempts at security
- Some run a business model that requires ignoring security
- Some say techology-X-will-save-us, but when X-fails-to-compile-perl, they scurry away in silence
- NIH syndrome remains strong

Limiting choice is related to safety

Safety System Innovations are crucial

- Old solutions may not match modern
- World market for computers now exceeds 5
- Don‘t get tied down by policy of backwards compatibility forever
- Innovation is crucial to the success of any technical endeavour
- Don‘t sleep on shoulders of giants



Questions?

Postel‘s maxim is dangerous

IETF: draft-thomson-postel-was-wrong-02.txt

In network protocols, there are evolutionary risk factors due to non-strict specification

Similar problem with conserving ABI+below, which benefits attackers greatly

POSIX (standard) is already too liberal

POSIX (implementations) are full of historical defacto-standard behaviours which attackers can rely upon