Pledge: where did it come from?

Was pledge invented in a light happy dream?

> „ *We stood beneath an amber moon*
> *Where hearts were entertaining June*
> *And softly whispered "someday soon"*
> *We kissed and clung together* “
> *…*

No, it is the outcome of nightmares.

My nightmares

- When Good Instructions go Bad: Generalizing return-oriented programming to RISC
  - Buckanan, Roemer, Shacham, Savage. 2008.

- Hacking Blind (BROP)
  - Bittau, Belay, Mashtizadeh, Mazières, Boneh. 2014.

ROP – Return Oriented Programming

Hijack control-flow with false return frames, running **gadgets**, combining artifacts effects

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

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

Also JOP, SROP, etc.

BROP – Blind ROP

An address-space oracle

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

Then uses various ROP methods.

(Large) software will never be perfect

<u>**Erroneous condition logic fails, then cascades through successive failures, often externally controllable**</u>

Results in illegal access/control of the program & libraries, or toying with kernel surface

Attacker tools and knowledge are improving, faster than developers can cope

I work on mitigations

Mitigations are inexpensive tweaks which impact attack methods – trying to diminishing their effectiveness

Some mitigations expose use of un-standardized behaviours

Defect detected —> **Fail Closed**

Pressure towards **robustness** in software.

Robust (adj.)

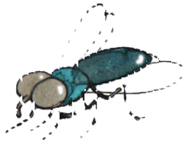
> ” When used to describe software or computer systems, *robust* can describe one or more of several qualities:
> - a system that does not break down easily or is not wholly affected by a single application failure
> - a system that either recovers quickly from or holds up well under exceptional circumstances
> - a system that is not wholly affected by a bug in one aspect of it “

On the way to the lush valley of *robust*, we must first cross the wilderness of fail-closed. We haven‘t finished that journey yet.

How to measure a good mitigation?

- Diminishes effectiveness of specific attack method
- Efficient, low overhead
- Easy to understand
- Easy to incorporate into old & new code
- One mitigation need not fix ALL the problems – let‘s hope they cooperate like aspirin + hot toddy
- Rise of a cult of followers & adopters also counts as a measure of success

Components attackers use



Knowledge	+	Mechanism	+	Objects
Substantial consistancy		Code Reuse		Filesystem
Location of objects (relative and absolute)		Syscalls		open fd‘s
Gadgets, constants, pointers, regvalues, etc.				

17 years of mitigation work

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 now)

Heretic! BSD was already perfect!

- The rules of engagement changed.
- Security concerns were not on the radar 30 years ago.
- Ignoring problems doesn‘t make them go away

This is research:

Discover & design new improvements, use base+ports to validate effective patterns

Earlier mitigations often need uplift

Example: ASLR

1. Randomize DSO bases… 2001
2. Randomize DSO order… 2003
3. Guard zones between.. 2005
4. Guard bottom of stack… 2017
5. Randomize internal objects.. 2017

...

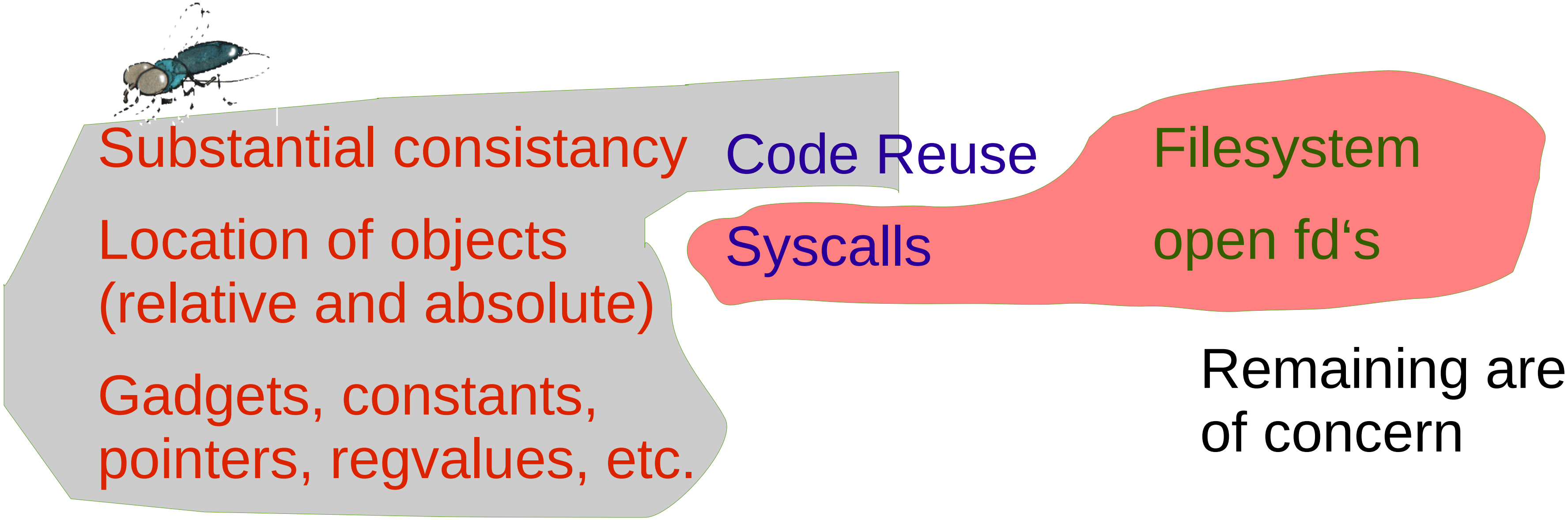
Mitigation Strategies

- Reduce externally-discoverable knowledge
- Improve historical weaknesses of permission models
- Disrupt non-standard control-flow methods
- Educate increasing use of fork+exec privsep

But not enough: if control is grabbed, **syscalls get used to act upon resources.**

Components attackers use

Knowledge + Mechanism + Objects



Substantial consistancy

Location of objects (relative and absolute)

Gadgets, constants, pointers, regvalues, etc.

Code Reuse

Syscalls

Filesystem

open fd‘s

Remaining areas of concern

Largely migitated... or works ahead

Where mitigations apply

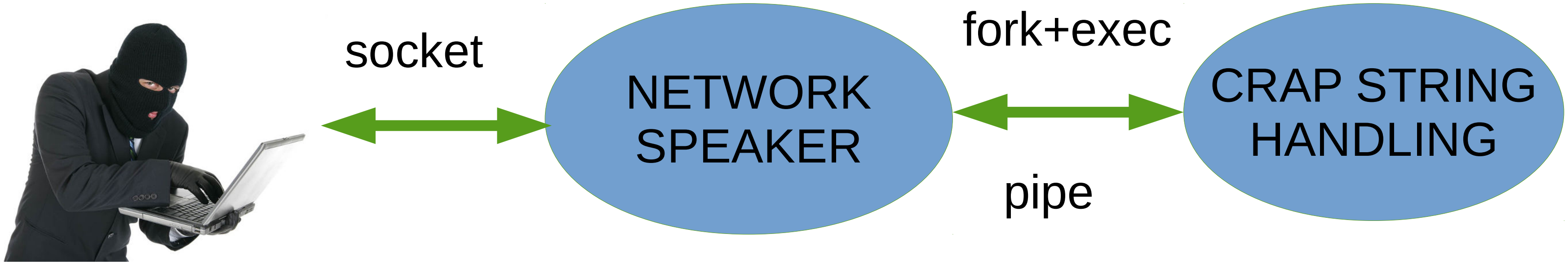
Stackoflow	→	ROP →	BROP
		privsep →	fork+exec
SSP	W^X	privdrop	pledge
hand-audit	ASLR		RETGUARD4
	Per-DSO SSP		per-DSO relink
	Stackghost		X-only .text
			poly-ret scrub?

Privsep + pledge

Stackoflow	→	ROP →	BROP
		privsep →	fork+exec
SSP	W^X	privdrop	pledge
hand-audit	ASLR		RETGUARD4?
	Per-DSO SSP		per-DSO relink
	Stackghost		X-only .text
			poly-ret scrub?

Privilege Separation

Many OpenBSD programs were rewritten to follow a design pattern called **Privilege Seperation** – Work domains are split into seperate processes.



socket

NETWORK SPEAKER

fork+exec

pipe

CRAP STRING HANDLING

Seperate security domains, in theory...

Pledges are POSIX subsets

Each pledge request allows a (carefully selected) subset of POSIX functionality

Subsets such as: **stdio rpath wpath cpath fattr inet dns getpw proc exec** …

Deep functional support in the kernel; much more than ¨seccomp¨ macros

Pledges are POSIX subsets

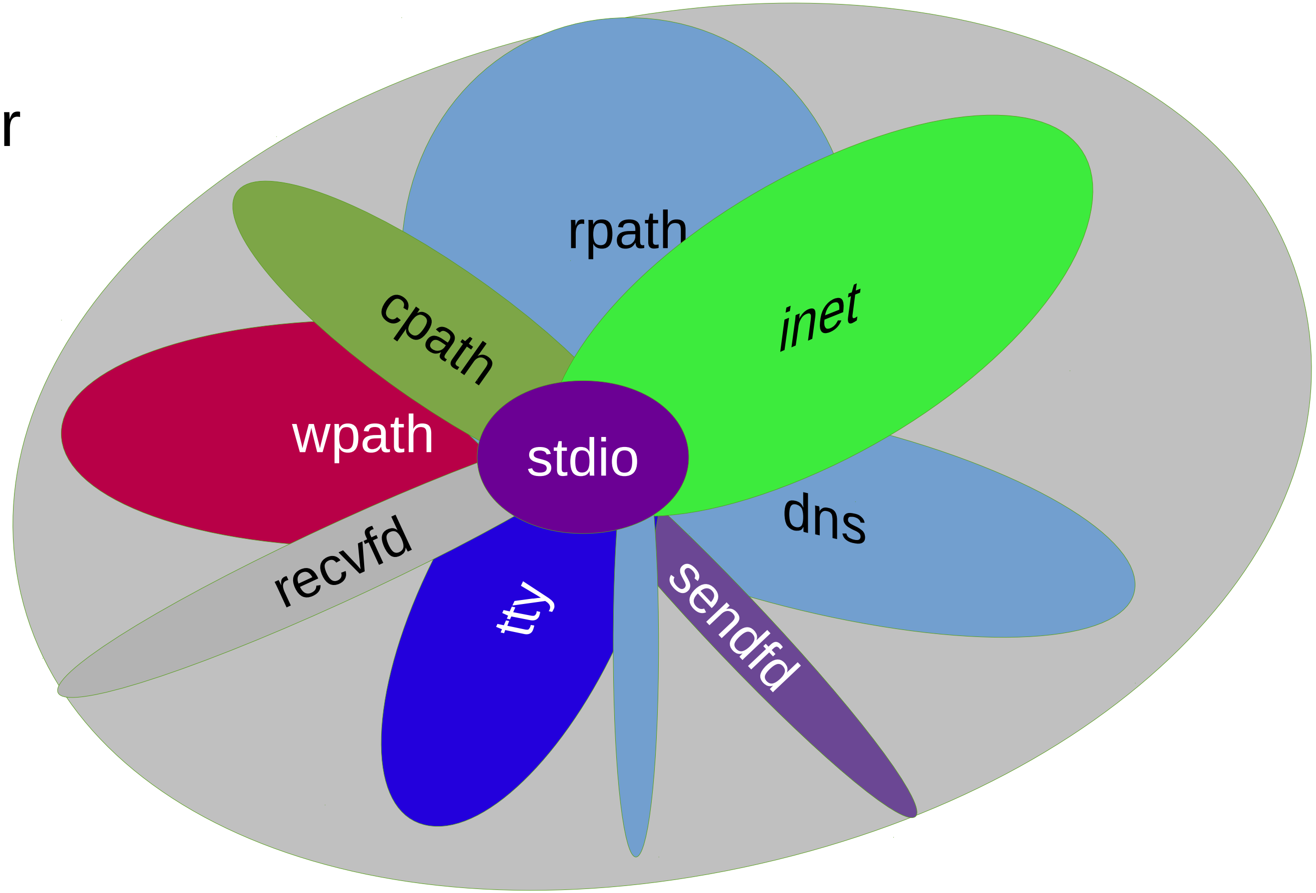
No subtle behaviour changes

No error returns

Fails-closed

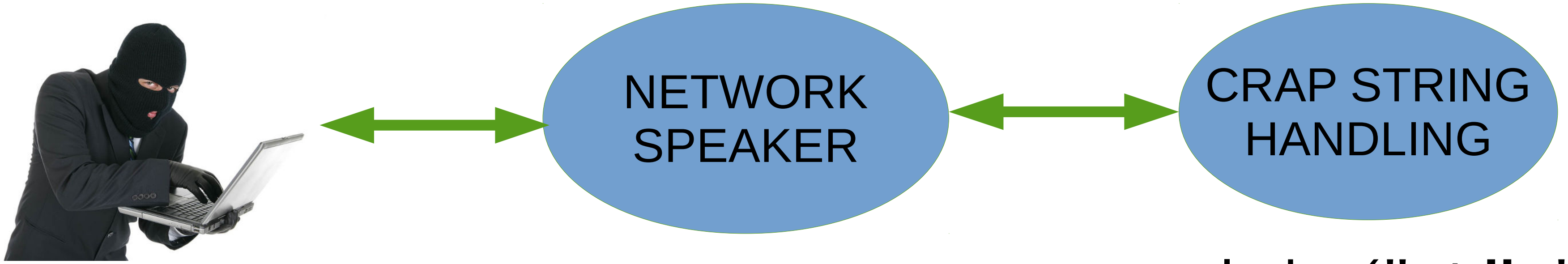
Illegal operations crash

Easy to learn



Privilege Separation + Pledge

Pledge ENFORCES the security-specialization of each process



NETWORK SPEAKER

CRAP STRING HANDLING

pledge("**stdio inet**")

pledge("**stdio**")

That wasn‘t so hard. Any questions?

How does pledge help privsep?

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

No behaviour changes, only detection of rule violation

Consider: **#define pledge(x,y) 0**

Shell-friendly

- Many programs are nominal ″shells″ -- spawn commands
- Ignoring this requirement leaves them **unprotectable**
- **proc** and **exec**, permit fork/execve related operations
- execve() turns off pledge features -- anticipates new image will enable pledges it needs
- If you don‘t use **exec**, it cannot bite you
- OpenBSD sh cannot open sockets. capsicum has no solution for this problem.

Hoisting – Handling Disappointment

- On occasion, pledge rules are extensive — exposing breadth of system call use by program
- **Hoisting** is the process of identifying initialization code which gets run late, and moving it early
- Refactoring results in programs with tighter pledge

- Depends on zeal of the developer...

pledgepath() — WIP

- Filesystem containment mechanism in development
- Pre-register required ffilepaths, dirpaths
  - vnode references grabbed, and rediscovered later by namei
- Like chroot in reverse?
- Decision between various TOCTOU scenarios – selecting a **fail-closed** behaviour of course

Developers, developers, developers!

Use of pledge in a program is always less complicated than the program itself!

Cannot pledge firefox due

- lack of inherent privsep
- fails to isolate syscall reach into different modules
- so everything must be allowed

chrome was strongly pledged in <1 week

- Google wrote it privsep from the start

OpenBSD Foundation

Thank you to all who support OpenBSD work through contributions to the OpenBSD Foundation

Remember – Pledge early, pledge often!