Effective Memory Safety Mitigations

@chrisrohlf

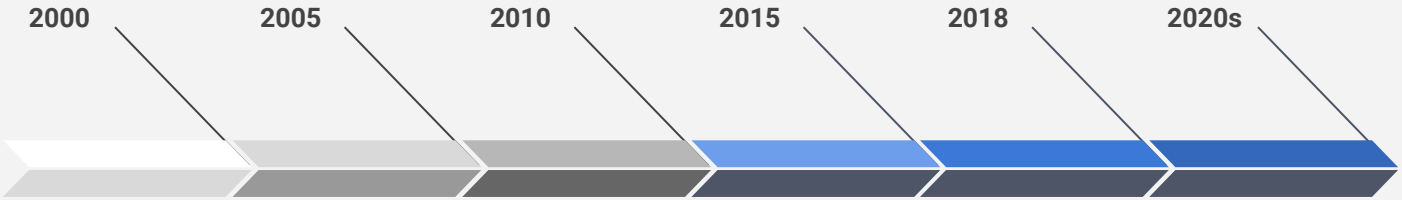
Agenda

History
Taxonomy
Exploit Mitigations
Custom Mitigations
Exploit Mitigation Effectiveness

Native Code Security

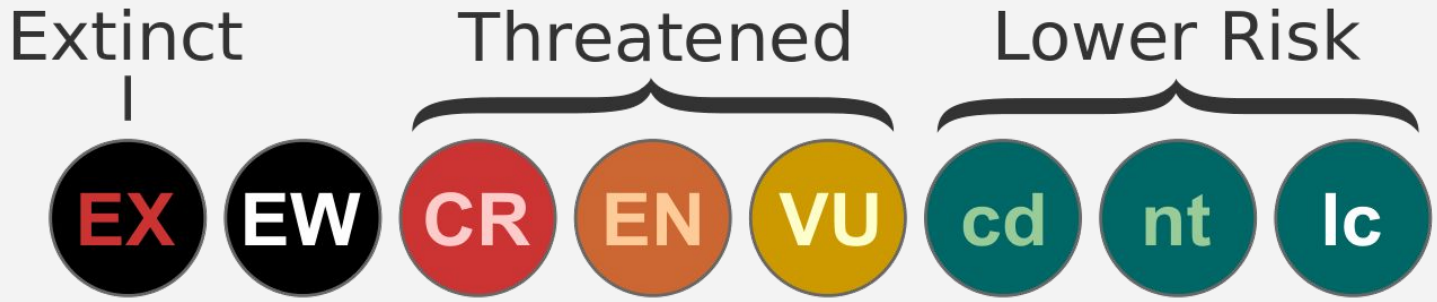
- C/C++ are heavily used in widely distributed software
- Tools, compilers, and libraries have all gotten better
- Memory safety vulnerabilities are still a challenge

Exploits and Mitigations



2000	2005	2010	2015	2018	2020s
Morris Worm	C++ vtable hijacking	Refinement of browser exploit techniques	Program specific techniques dominate	Sandbox escapes are required for any serious target	Hardware assisted CFI/CFG/RFG
8LGM	Browser Exploits	Memory disclosures	ASLR bypass is required	Exploit chains require multiple steps	Immutable containers, and sandboxes become the default everywhere
NCSA HTTPD	Heap Spray	Flash techniques	JIT era begins	Out of process JIT appears	CFI/CFG bypass required
SplitVT	Heap metadata (unlink, voodoo malloc tricks)	The end of an era for general exploit techniques	Shellcode is dead, DEP is everywhere	Side channels break everything	RAP (?)
Smashing the Stack	Borrowed Code Chunks	ROP (refinement of earlier ret2text)	Sandbox era begins	Kernel protections become the default	Shadow process memory verification
NX Stack	PAX		Mobile jailbreaks begin to gain influence	Execute only memory	Out of process components
Stack Canaries	ASLR		CFI/CFG appears		
Ret2libc					
StackGuard					
Heap Overflows					
Format Strings					

This is an incomplete timeline. Consult your local hacker historian for more information



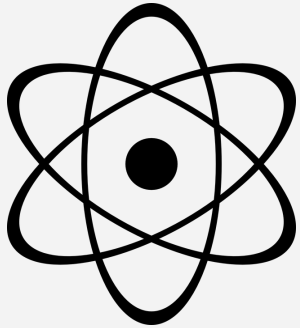
Format Strings
Kernel NULL Ptr Deref
NOP Sled to Shellcode

Stack Overflows
Vtable Hijacking
Double Free
new implicit overflow
JIT Spray
JOP
Heap Unlink
GOT Overwrite

Memory Disclosures
Use After Free
Heap Overflows
Integer Overflows
Side Channels
Uninitialized Memory
Off By One
TOCTOU
Double Fetch
Type Confusion
ROP/BROP
COOP
Heap Spray/Feng Shui

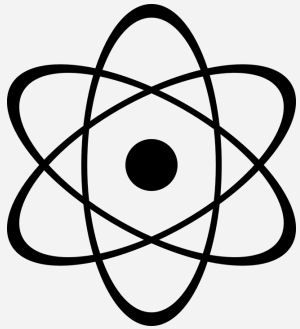
Exploit jargon is misleading and inaccurate

Exploit Taxonomy



- Programs are a collection of intended state transitions
- Exploit techniques chain together primitives to add unintended state transitions to a program
- Effective mitigations place constraints on all state transitions to ensure they are valid and intended
- Program design and languages can work against the constraints exploit mitigations are designed to impose

Exploit Taxonomy

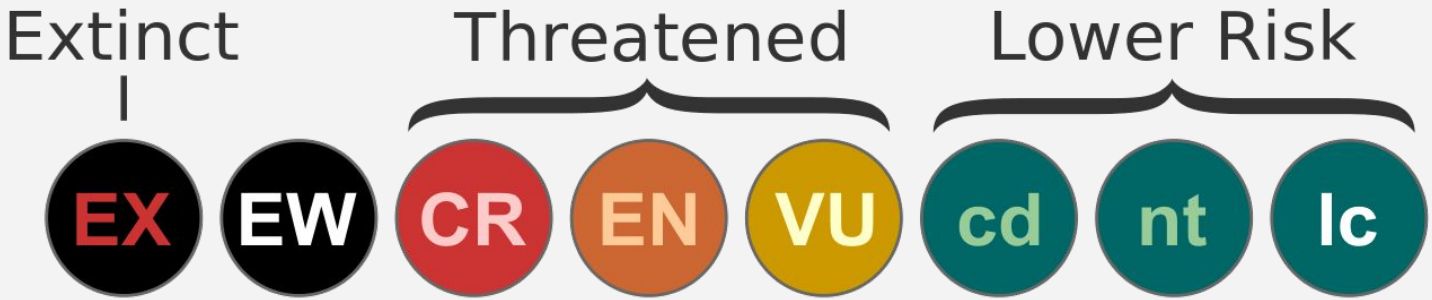


- Primitive operations that modify program state: RWX

- All vulnerabilities can be classified according to the primitives they allow an exploit to control or influence
  - Not all primitives useful to an exploit require unintended state transitions, precise heap grooming is an example

- An effective exploit mitigation reduces the influence or control of these primitives despite the vulnerability being present

- Defeating an exploit mitigation requires control of a primitive the mitigation does not impose a constraint on



Read

Write

Execute

Exploit conservation status: Least Concern

Exploit State Transitions

- Write ➡ Execute
  - An arbitrary write grants execute
  - Exploit: Overwriting a saved return address
  - Mitigation: Stack canary detects the overwrite
    - Relative write can give us a relative read

- Read ➡ Write ➡ Execute
  - Write may require a read
  - Exploit: Overwriting a function pointer in the heap
  - Mitigation: ASLR introduces a read primitive requirement
  - “Where are my ROP gadgets mapped?”

Exploit State Transitions

- Relative primitive allows for read/write of some number of bytes relative to a base address
- Absolute primitive allows for arbitrary read/write of some number of bytes at an arbitrary address

Exploit State Transitions

- Absolute write

```
attacker_address = attacker_value;
```

- Relative write

```
string_pointer = malloc(1024);
string_pointer[attacker_offset] = attacker_value;
```

- How can we use the relative write to achieve relative read?

Effective Exploit Mitigations

- Introduce constraints between unintended state transitions
- Reduce reliability by removing predictable conditions
- Low cost, or invisible to software engineers and users
- Provided by a compiler, OS runtime, or library
- May be program design specific
- Sometimes tied to hardware
- Effectiveness is inversely proportional to program complexity

Mitigations Deny Unintended State Transitions

ASLR/PIE (rw)
Heap Isolation (rw)
Guard Pages (rw)
Out of Band Metadata (rw)
Meta-Data Canaries (w)
RELRO (w)
NX/DEP (x)
CFI/CFG (x)
Delayed Free (rwx)

Mitigations By Program Design

- Sandboxes mitigate successful exploits by
  - Separating them from sensitive resources
  - Reducing attack surface for further privilege escalation

- Microsoft Edge JIT Process Server
  - ACG places dangerous RWX allocation and native code compilation into a separate process
  - Doesn’t protect against Incorrect JIT Code Emission

iOS Mobile Safari JIT Allocations

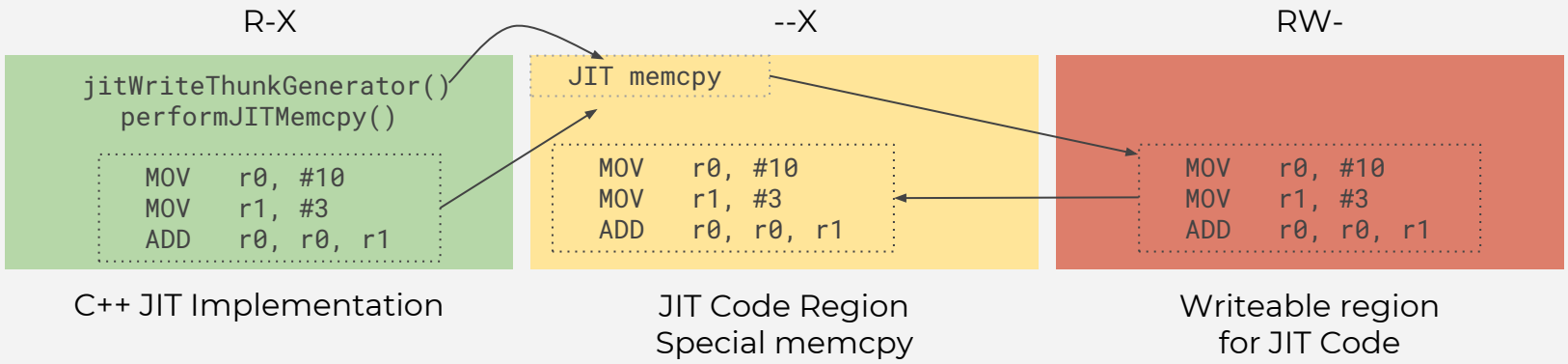
- Javascript JIT compilers remove the constraints imposed by NX/DEP by needing memory that transitions from W to X

- ARMv8 allows for execute only memory

- Safari creates two virtual mappings to the same physical JIT memory region using `mach_vm_remap`
  - One mapping is read/write, the other is executable
  - The address of the writeable address is ‘secret’

iOS Mobile Safari JIT Allocations

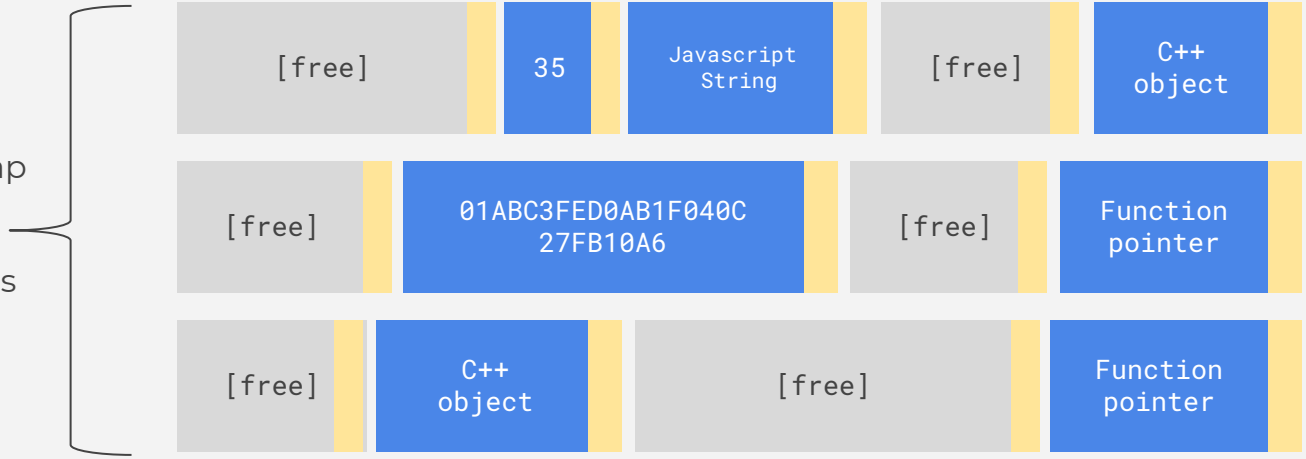
- The JIT emits a custom memcpy function that has the address of the writeable region encoded as a constant
  - The address of the writeable region is wiped and forgotten from the JIT
- At runtime the JIT uses the special memcpy to write code to the writeable region
- The execute-only mapping mirrors the writeable mapping
- This removes the RWX mapping that was previously used for exploitation



General Purpose Heap Allocator

- Your typical heap allocator doesn't understand object types
- You are probably storing untrusted program inputs adjacent to sensitive control flow metadata

Example system heap maintained by malloc/free or new/delete operators



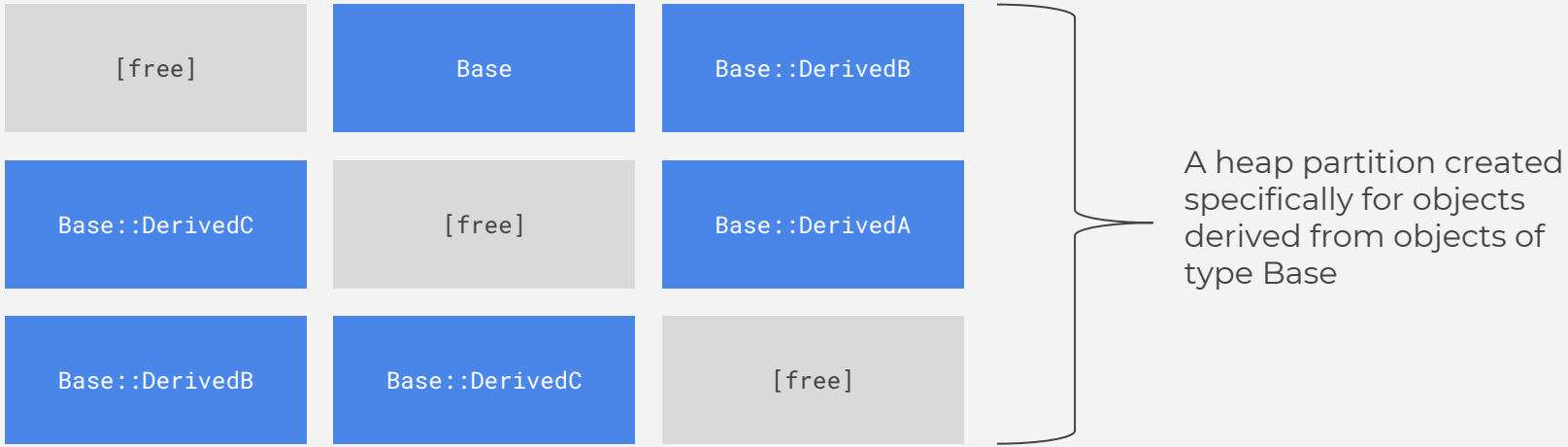
Heap Isolation

- Heap Isolation is an effective exploit mitigation
  - Separates allocations by type and/or size
  - Can be used to restrict read/write
  - Let me tell you about RenderArena
    [webkit-dev RenderArena: Teaching an old dog new tricks - 11/13/2012]

- Partition Alloc was introduced in Chrome to isolate objects typically used in DOM Use After Free exploits

Heap Isolation / PartitionAlloc

- Heap isolation solves this by storing objects using type or size
- Metadata is always stored out of band
- Consider a set of C++ objects, each derived from **Base**
- A partition can be allocated only for objects derived from **Base**
- Vulnerabilities that manipulate the heap are harder to exploit



Hardened PartitionAlloc

- Partition Alloc has strong security guarantees by default
- Hardened PartitionAlloc introduced more
  - Free list randomization
  - Freelist entries are randomly selected upon allocation
  - Allocated slots are surrounded by a canary value that is unique per-partition and XOR'd by the last byte of its address
  - New allocations are initialized with 0xDE
  - Freelist pointers are checked for a valid page mask and root inverted self value
  - Delayed free of all user allocations using a vector stored with the partition root
  - Free'd allocations have their user data memset before they're added to the delayed free list
  - Better double free detection

Brute Forcing With Fork

- Forking daemons provide an opportunity to defeat secrets like ASLR and Stack Canaries by brute force

- Fork enables BROP to extract closed source binaries over the network for further inspection

- Partial Solution: Fork Guard
  - Designed to reduce the reliability of (B)ROP by removing unused code from memory
  - Works on closed source programs

ROP / BROP

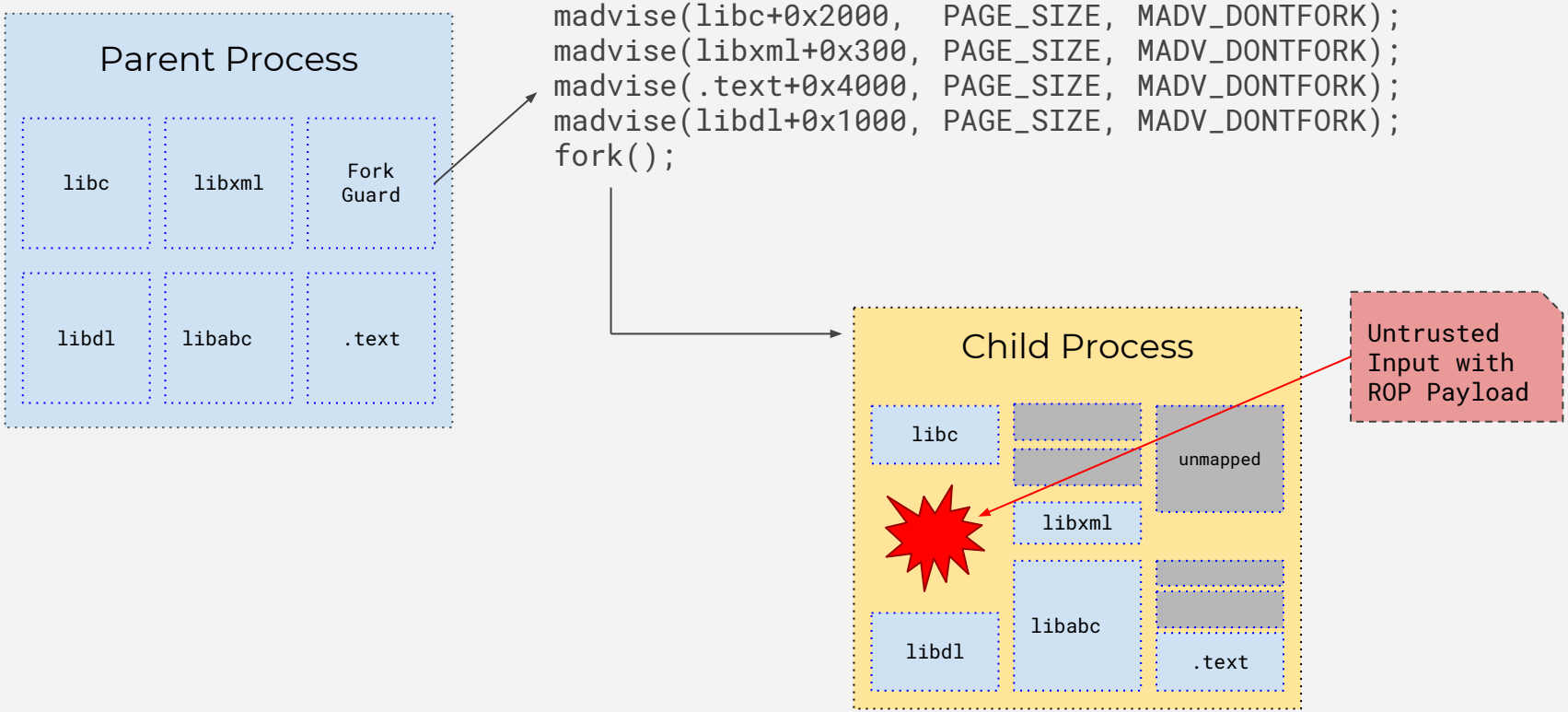
- ROP reuses existing code pages to stitch together payloads

- BROP is a technique for exploiting stack overflows in forking daemons for which you have no binary
  - Brute force the stack canary byte by byte
  - Iterate on stored return address until you can call `write` to leak the binary over the existing network socket
  - Noisey exploit technique even when it works
    - Only needs to work once to get the binary for further reverse engineering

Fork Guard

- `LD_PRELOAD = forkguard.so`
- Loads a whitelist of functions the child process needs
- Discovers all code pages in the process via dynamic linker introspection
- Overloads libc `fork` and uses `madvise` with `MADV_DONTFORK` to tell the kernel which pages can be dropped in the child process
- Tracing mode is used to expand the initial whitelist by monitoring child process signals from attempts to execute code in unmapped pages

Fork Guard


Parent Process
libc
libxml
Fork Guard
libdl
libabc
.text
madvise(libc+0x2000, PAGE_SIZE, MADV_DONTFORK);
madvise(libxml+0x300, PAGE_SIZE, MADV_DONTFORK);
madvise(.text+0x4000, PAGE_SIZE, MADV_DONTFORK);
madvise(libdl+0x1000, PAGE_SIZE, MADV_DONTFORK);
fork();
Child Process
libc
unmapped
libxml
libabc
libdl
.text
Untrusted Input with ROP Payload

Fork Guard

- Results in less code for (B)ROP payloads to be constructed from
  - Naive implementation resulted in %35 less code pages in Nginx
- Reduces the reliability of existing (B)ROP payloads
- Effective in forking daemons whose child processes deal with untrusted input
- Works on closed source binaries
- Small performance penalty when the parent process first calls `fork`

Program Design and Exploit Mitigations

- ASLR strength is highly dependent on program design
  - ASLR in a 32 bit browser provides no value
  - ASLR in a 64 bit browser provides some value
  - ASLR in a forking daemon is defeated by brute force
  - ASLR in standalone programs with no scripting environment provides a lot of value and is not easily bypassed

- DEP is highly dependent on ASLR
  - ROP provides an easy alternative to shellcode
  - Without ASLR ROP payloads are trivial to construct

Program Design and Exploit Mitigations

- KASLR less useful than ASLR

- Browser designs actively subvert simple mitigations
  - Untrusted arbitrary computation of Javascript means all state transitions are intended by design
  - Javascript JIT engines, 20+ parsers, plugins
  - Resource constraints via sandboxing is required

- ASLR and Stack canaries easily defeated in forking daemons

- Any program with PCRE allocates RWX memory

Languages and Exploit Mitigations

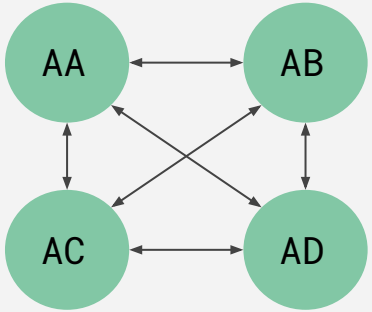
- C++ introduces a number of challenges for exploit mitigation development

- Polymorphic classes introduce indirect control flow as part of its core design
  - vtable pointers

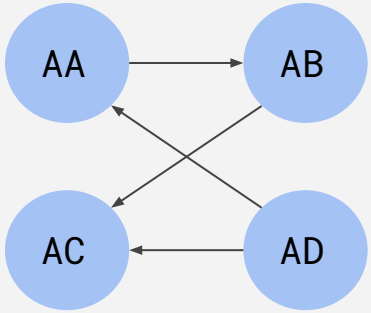
- Overloading of operators that introduce implicit operations such as reference counting and memory management

CFI/CFG

- Coarse grained CFI (naive)
  - The CFG is a set of valid addresses
  - Only a partial restriction of execute primitive
- Fine grained forward edge CFI (strict)
  - The graph is enforced explicitly according to intended code flow
  - C++ indirect calls are a challenge, function signatures are required
- Fine grained backward edge CFI
  - Restricts which function a function can return to
  - Usually a Shadow Stack, subject to various attacks

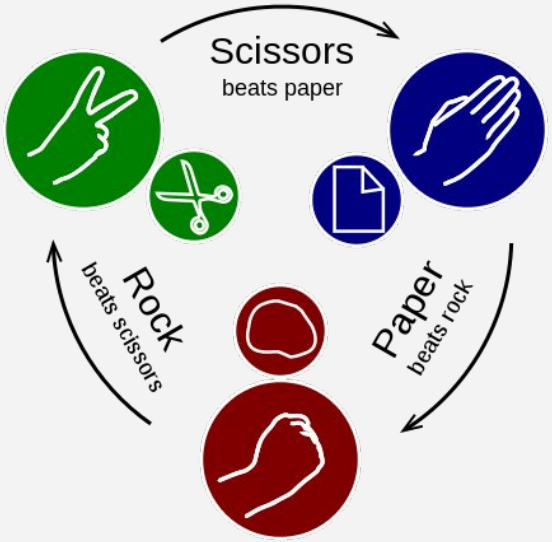


Coarse Grained CFI



Fine Grained CFI

Exploit Mitigation Effectiveness



- Mitigations become more effective when combined together to form a mesh
  - ROP defeats DEP but requires an ASLR bypass
  - Memory disclosures can defeat ASLR but are mitigated by memory isolation strategies

- What makes CFI/CFG so strong?
  - Enforces the intended control flow of the program
  - Not as strong without type checking and pre-computed order of operations

Challenges

- New exploit mitigations are often reactive, and can take several years before widespread adoption
  - New exploit techniques are adopted immediately

- New code often reintroduces the problems of the past

- Engineering time trade offs
  - Google NaCl - Secure design but engineering investment too high
  - Isolated Heaps - Memory allocation APIs add complexity

- Performance trade offs
  - Full CFI - Can slow down a process by a non-trivial amount
  - ASLR - Almost no measurable performance impact on modern hw

Thank you for listening