7 Dumb Ways to Bypass Syscall Filters

Brian Welch

About me

- In between developer gigs
- Find my projects at https://brianwel.ch

Disclaimer

- Only talking about Linux
  - I know nothing about syscall filtering on Windows
- Mainly focusing on the amd64 architecture
- Mainly focusing on `seccomp-bpf` as the syscall filter mechanism
- Most of my testing was done on Ubuntu 20.04 (Linux kernel 5.4)

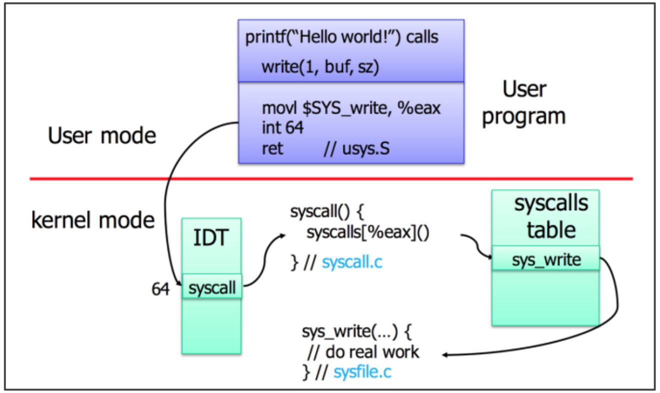
Agenda

- Brief intro to syscalls/filters/seccomp
- Overview of seccomp filter bypasses (increasing in complexity)

Background

What are Syscalls?

- The main interface between userland processes and functionality in the kernel
- The functions you call in C programs like `open`, `read`, and `write` are libc wrappers around syscalls of the same name
- Syscalls are identified by number (which changes based on the architecture)



```
printf("Hello world!") calls
  write(1, buf, sz)
```

```
movl $SYS_write, %eax
int 64
ret      // usys.S
```

User program

User mode

kernel mode

IDT

64 syscall

```
syscall() {
  syscalls[%eax]()
} // syscall.c
```

syscalls table

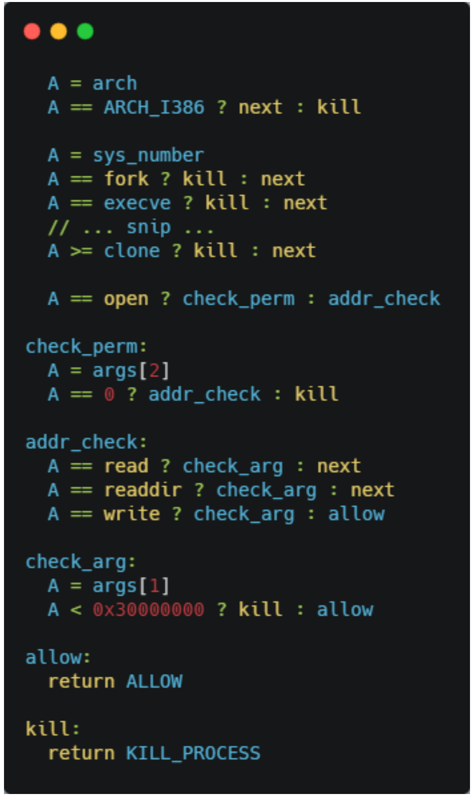
sys_write

```
sys_write(...) {
  // do real work
} // sysfile.c
```

Useful way of thinking about the userland-syscall-kernel boundary (source)

What are Syscall Filters?

- There’s a few ways a syscall filter can be implemented
  - Historically, some people rolled their own via `prctl`
  - Great talk: Escaping the (sand)box from CONFidence 2017
- State of the art: `seccomp-bpf` and `libseccomp`
  - See here and here for relevant LWN articles
  - Compiled BPF program that executes in the kernel each time a syscall is requested
  - Allows for prohibiting syscalls and their argument values
- How do we bypass these?
  - Find flaws in the implementation of the syscall filter
  - Find flaws in the set of permissible syscalls

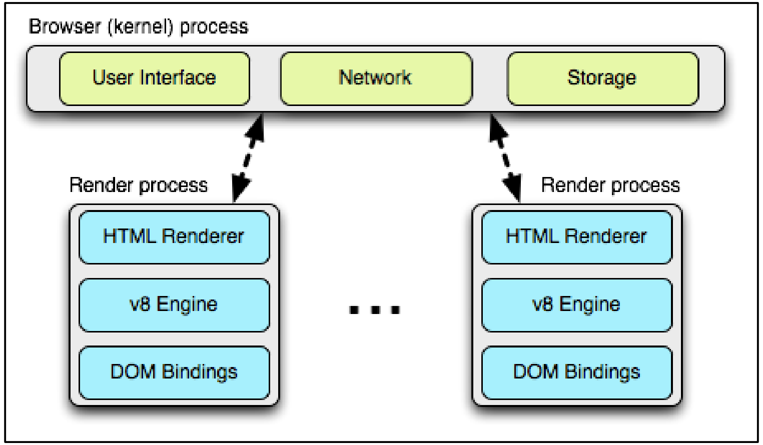


```
  A = arch
  A == ARCH_I386 ? next : kill

  A = sys_number
  A == fork ? kill : next
  A == execve ? kill : next
  // ... snip ...
  A >= clone ? kill : next

  A == open ? check_perm : addr_check

check_perm:
  A = args[2]
  A == 0 ? addr_check : kill

addr_check:
  A == read ? check_arg : next
  A == readdir ? check_arg : next
  A == write ? check_arg : allow

check_arg:
  A = args[1]
  A < 0x30000000 ? kill : allow

allow:
  return ALLOW

kill:
  return KILL_PROCESS
```

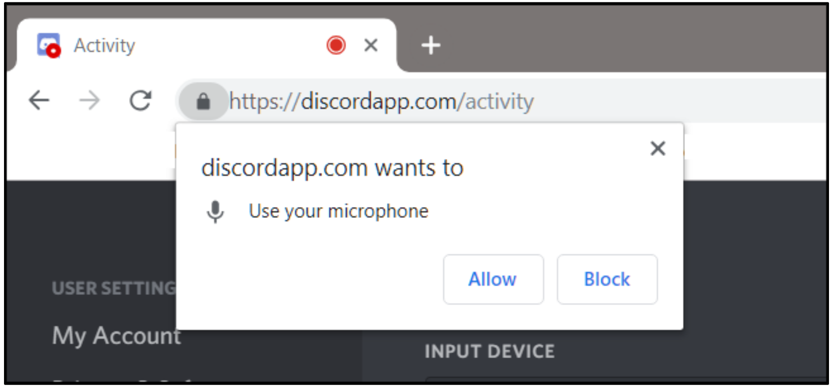
Example of a syscall filter implemented in the seccomp-tools metalanguage for a 32-bit x86 program

Defense in Depth: Syscall Filtering

- Defending against adversaries that already have code execution
- Overall goal is to limit the kernel attack surface
- Browsers are useful case study for syscall filtering
  - Helpful resource: Chromium’s Linux sandbox design document



Simplified view of Chrome’s multi-process architecture (source)



discordapp.com wants to

Use your microphone

Allow

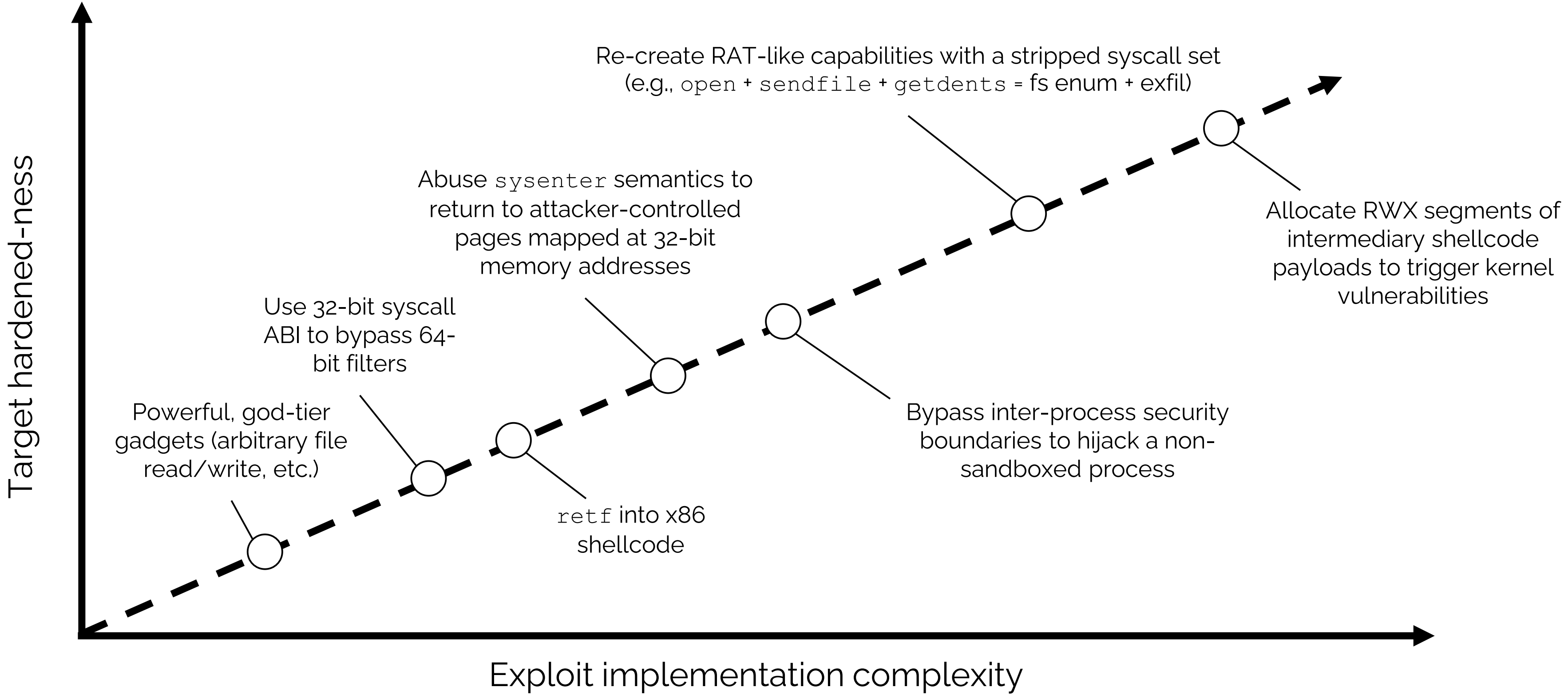
Block

Example of granular access to system resources in the Chromium security architecture

Why Do We Need to Understand This?

- Syscall filters are the new CTF meta
  - Overwriting `__free_hook` with a one-gadget is old news
- Systems are becoming increasingly hardened; we need to stay ahead of the curve
  - Point and click access opportunities without advanced protections are a dying breed
  - Limited-execution environments guide us towards system exploitation rather than single-process exploitation

The Spectrum of Syscall Filter “Bypasses”


Target hardened-ness
Exploit implementation complexity
Powerful, god-tier gadgets (arbitrary file read/write, etc.)
Use 32-bit syscall ABI to bypass 64-bit filters
retf into x86 shellcode
Abuse sysenter semantics to return to attacker-controlled pages mapped at 32-bit memory addresses
Bypass inter-process security boundaries to hijack a non-sandboxed process
Re-create RAT-like capabilities with a stripped syscall set (e.g., open + sendfile + getdents = fs enum + exfil)
Allocate RWX segments of intermediary shellcode payloads to trigger kernel vulnerabilities

(Relatively) Easy Workarounds

Powerful Syscalls and Easy Wins

- Some syscalls (`fork`, `execve`, etc.) have the side effect of (sometimes) completely removing the current process's syscall filter
  - This only works for poorly-implemented `ptrace`-based filters; you're mostly safe if you use `libseccomp`
- Fortunately, some sane protections exist:
  - `seccomp-bpf` programs loaded in the kernel are preserved across `execve` and `fork` calls
- Other very dangerous syscalls:
  - Arbitrary arguments to `open`/`read`/`write`: File read/write, procfs shenanigans, etc.
  - `fork`/`ptrace`: Intercept and manipulate syscalls on their way in to the kernel (example here)
  - `process_vm_readv` and `process_vm_writev`: Access to other processes' address spaces

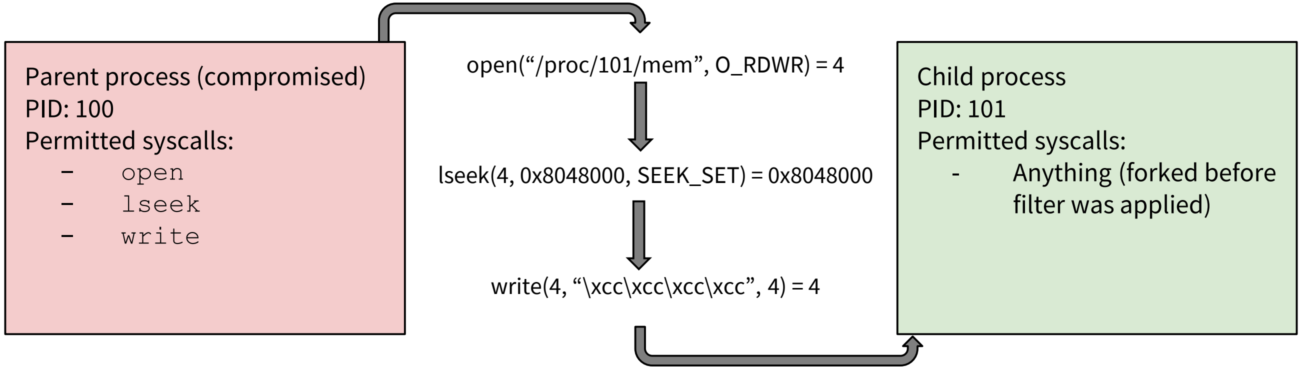
32-bit ABI Hacks

- When executing 64-bit x86 instructions, you can still execute syscalls using the 32-bit syscall ABI (i.e., via `int 0x80`)
    - These syscall numbers are *different* than their 64-bit ABI equivalents (bitwise ORed with 0x40000000)
    - Remember that seccomp filters are implemented by checking for syscall number equivalence
- Similar (but different) technique involves switching the processor into 32-bit mode
    - Syscall numbers in 32-bit x86 mode are completely different than their 64-bit equivalents
    - Do this by executing a far return (`retf`) instruction with the stored CS register set for 32-bit mode
    - This can also be triggered when returning from a `sysenter` instruction issued in 64-bit mode
- This is why you shouldn’t implement your own filters
    - `libseccomp` (generally) takes care of these architecture edge-case details

Entering the Arcane

Overwrite Code with `/proc/{PID}/mem`

- Exploit a sandboxed parent process → modify an un-sandboxed child process
  - Does require the ability to `lseek` around the opened pseudofile
- Can also modify your own process with `/proc/self/mem`
  - Not necessarily as useful, since you're still beholden to the same syscall filter
- Writing to `/proc/{PID}/mem` ignores the mapped permissions!
  - Relevant CTF challenge: writeonly from GoogleCTF 2020



Parent process (compromised)
PID: 100
Permitted syscalls:

- `open`
- `lseek`
- `write`

open("/proc/101/mem", O_RDWR) = 4

lseek(4, 0x8048000, SEEK_SET) = 0x8048000

write(4, "\xcc\xcc\xcc\xcc", 4) = 4

Child process
PID: 101
Permitted syscalls:

- Anything (forked before filter was applied)

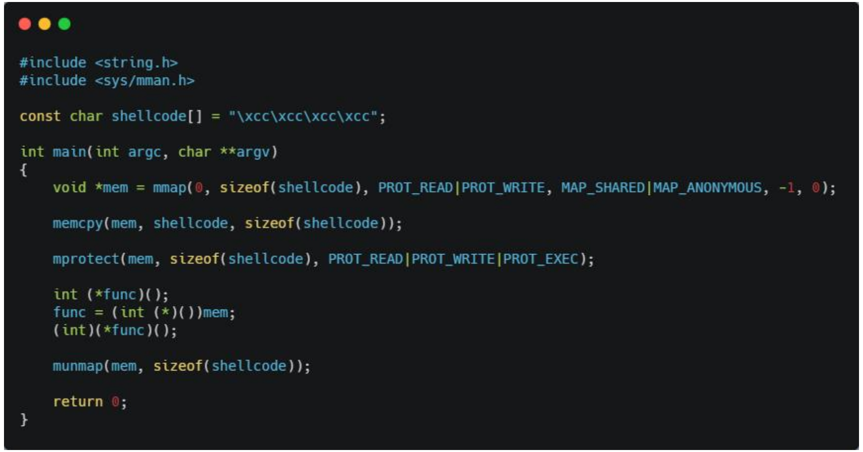
Making the Most of What You Have

- Sometimes, a highly restrictive filter is still sufficient to do what we want
  - Do we really need *arbitrary* code execution?
- Minimal file system enumeration and file retrieval:
  - Open files via `open` or `openat`
  - Enumerate directory entries from `open`-ed director file descriptors with `readdir` or `getdents`
  - Send files directly to a socket via `sendfile`
  - Assume that we can exfiltrate data using the socket we remotely exploited over
- Relevant CTF challenge from pbctf 2020 can be found here

RWX Staging for Follow-on Kernel Exploits

The Easy Ways: `mmap` and `mprotect`

- `mmap`: Syscall to create new memory mappings
- `mprotect`: Syscall to change the protections of existing pages of memory
- Most sane syscall blocklists would disallow these
  - Or, more specifically, disallow the `PROT_READ`|`PROT_WRITE`|`PROT_EXEC` argument



```c
#include <string.h>
#include <sys/mman.h>

const char shellcode[] = "\xcc\xcc\xcc\xcc";

int main(int argc, char **argv)
{
    void *mem = mmap(0, sizeof(shellcode), PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);

    memcpy(mem, shellcode, sizeof(shellcode));

    mprotect(mem, sizeof(shellcode), PROT_READ|PROT_WRITE|PROT_EXEC);

    int (*func)();
    func = (int (*)())mem;
    (int)(*func)();

    munmap(mem, sizeof(shellcode));

    return 0;
}
```

Example of a C harness program for changing the permissions of a page of memory to RWX and executing x86 shellcode stored there (adapted from here)

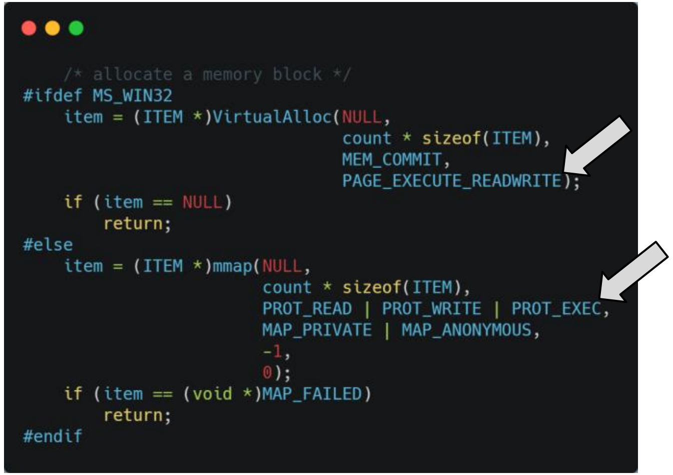
`SYS_ipc` and Friends

- Set of syscalls for managing inter-process shared memory segments
  - On 32-bit x86, `SYS_ipc` is the “gateway” syscall
- Can be used to allocate RWX pages at arbitrary addresses without the use of `mmap/mprotect`
- Easily setup in a ROP chain with 2 syscalls:
  - Create a shared memory region via `SHMGET` operation with `IPC_CREATE|0o777` flag
  - Use the returned `key_t` to load the region into the current process, using the `SHM_EXEC` flag to mark as executable
- Example usage [here]()

Screenshot from inside your computer when you map RWX memory via `SYS_ipc`

Side Note: You Don’t Always Have to BYORWX

- Scripting languages (especially jitted ones) love to allocate RWX segments
- Up until recently, WebAssembly programs in Blink and WebKit were loaded into RWX segments



```
    /* allocate a memory block */
#ifdef MS_WIN32
    item = (ITEM *)VirtualAlloc(NULL,
                                count * sizeof(ITEM),
                                MEM_COMMIT,
                                PAGE_EXECUTE_READWRITE);
    if (item == NULL)
        return;
#else
    item = (ITEM *)mmap(NULL,
                        count * sizeof(ITEM),
                        PROT_READ | PROT_WRITE | PROT_EXEC,
                        MAP_PRIVATE | MAP_ANONYMOUS,
                        -1,
                        0);
    if (item == (void *)MAP_FAILED)
        return;
#endif
```

Code from the CPython `ctypes` module (source)

Wrapping Up

Lessons Learned

- Use syscall allowlists instead of blocklists
- Use hardened libraries (i.e., `libseccomp`)
- Syscall filtering: should only be one wall in your layered defense

Future Learning Resources

- Read Linux kernel source code

Thanks for your time

Reach out for slides and/or code samples