nccgroup
freedom from doubt

# Linux-Inject

**Tyler Colgan, NCC Group**

nccgroup
freedom from doubt

# Overview

- **What does it do?**
- **What can you do with it?**
- **Demos**
- **Limitations**
- **Comparison with Windows DLL injection**
- **Comparison with LD_PRELOAD**
- **How does it work?**

nccgroup
freedom from doubt

# What does it do?

- Loads a shared object into a running process
- Provides Linux analogue of Windows DLL injection
- Calls ptrace() on target process
- Essentially acts as a custom debugger
- Injects shared object loading code into the target process

nccgroup
freedom from doubt

# What can you do with it?

- Easily execute functions inside the context of another process
- Whichever function in the SO is marked with __attribute__((constructor)) will be called when the SO gets loaded
- So, you get code execution as soon as your SO is loaded!
- Ultimately, it lets you modify the behavior of any program you're allowed to attach to with ptrace()

nccgroup
freedom from doubt

# What can you do with it? (cont.)

- Create new threads
- Call exec() and run a different program
- Change memory permissions with mprotect()
  - Make code sections writable
  - Make data sections executable
- Output debug messages

nccgroup
freedom from doubt

# What can you do with it? (cont.)

- Hook functions
  - NOTE: You have to do this yourself in your shared object's constructor!
  - Easy way: unload competing shared object with dlclose()
  - Hard way: manually redirect function calls by modifying process memory
- Temporarily redirect functions at runtime for debugging purposes
- Video game hacking, just like on Windows
  - Anti-cheat technologies are currently much less developed on Linux

nccgroup
freedom from doubt

# Demos

nccgroup
freedom from doubt

# Limitations

- Must be able to call ptrace() on the target process
  - Does not let you attach to higher-privileged processes
  - Many systems disallow ptrace()ing non-child processes by default via /proc/sys/kernel/yama/ptrace_scope
  - If you can’t change this kernel setting, this approach simply can’t work
- Does not perform hooking, only injection
  - If you want to redirect functions, you need to do the dirty work of hooking yourself
  - However, you get code execution as soon as your shared object is loaded
  - More on this later

# Comparison with Windows DLL injection

- **VirtualAllocEx**
  - Performed from within the target process via injected malloc() call
- **WriteProcessMemory**
  - Performed by copying data from injector to target with ptrace(PTRACE_POKEDATA)
- **CreateRemoteThread**
  - Not needed, because we're hijacking a pre-existing thread
- **LoadLibrary**
  - Performed by injected call to __libc_dlopen_mode()

# Comparison with LD_PRELOAD

- **Advantages over LD_PRELOAD**
  - Targets processes that are already running
  - Does not require you to control the program's environment as it starts up
  - Not as easily detectable in the environment
  - (Admittedly, still not hard to detect)
- **Disadvantages compared to LD_PRELOAD**
  - Requires explicit function hooking in order to perform function redirection
  - Interrupts and temporarily redirects program execution, which can cause concurrency issues in large multi-threaded applications
  - (Still working on debugging and fixing various strange and intermittent crashes)

nccgroup
freedom from doubt

# How does it work?

1. Attach to target process with ptrace()
2. Inject loader code into target process
3. Target process executes the loader in several parts:
    a. Loader allocates memory with malloc() [VirtualAllocEx]
    b. Injector copies path on disk shared object to allocated buffer [WriteProcessMemory]
    c. Loader calls __libc_dlopen_mode() to load shared object [LoadLibrary]
    d. Injector checks whether shared object was loaded successfully
    e. Loader calls free() on the allocated buffer
4. Injector restores previous state and detaches from target process

nccgroup
freedom from doubt

# Attaching to target and injecting loader

- Requires /proc/sys/kernel/yama/ptrace_scope to be set to 0
- Injector finds an executable region of memory to store the loader
  - Reads /proc/[pid]/maps and takes the first region marked as executable
- Backs up whatever's stored there
- Backs up registers
- Copies loader in place and points program counter at it
- Continues target's execution in order to start running the loader

# Loader, part 1 – malloc()

- This step is analogous to calling VirtualAllocEx on another process on Windows
- Injector determines the address of malloc() inside the target process by reading its memory map
- Passes the malloc() addr  and the amount of memory to allocate to the loader via registers
- Need to allocate buffer in order to store full path on disk to the shared object we want to load
- Loader halts execution after malloc() returns
- Injector reads registers in order to see malloc() return value
- malloc() return value provides address of allocated buffer

nccgroup
freedom from doubt

# Loader, part 2 – Preparing to load

- This step is analogous to calling WriteProcessMemory on Windows
- The injector already has the address of the newly allocated buffer
- It uses ptrace() to copy the disk path of the shared object we want to load into the buffer
- Injector gets the address of __libc_dlopen_mode() and passes it to the loader via a register
- With this done, it continues the target process' execution

# Loader, part 3 – __libc_dlopen_mode()

- This step is analogous to calling LoadLibrary on Windows
- Loader calls __libc_dlopen_mode() to load the shared object
- It passes the previously allocated buffer as an argument so that __libc_dlopen_mode() knows where to find our shared object
- After __libc_dlopen_mode() returns, the loader returns control to the injector
- The injector checks the target's register values and /proc/[pid]/maps entries to check whether the shared object actually got loaded

# Loader, part 4 – Cleanup

- Injector continues the target's execution
- Loader calls free() on the allocated buffer and returns control back to the injector
- The loader's job is done, so restore the target process' state from before the injection
  - Overwritten memory
  - Register values
- Injector detaches from the target process and sends it on its merry way
- At this point, injection is (hopefully) complete!

# More info

- Source code
  - https://github.com/gaffe23/linux-inject
  - NOTE: This is a personal repo, so it's an unofficial release
  - Official release to come after BH
- Contact info
  - Tyler Colgan
  - tyler.colgan@nccgroup.trust

nccgroup
freedom from doubt



## North America

Atlanta

Austin

Chicago

New York

San Francisco

Seattle

Sunnyvale

## Europe

Manchester - Head Office

Amsterdam

Cheltenham

Copenhagen

Edinburgh

Glasgow

Leatherhead

London

Luxembourg

Milton Keynes

Munich

Zurich

## Australia

Sydney