Runtime Process Insemination

Shawn “lattera” Webb

SoldierX

https://www.soldierx.com/

Who Am I?

- Just another blogger
    - 0xfeedface.org
- Professional Security Analyst
- Twelve-year C89 programmer
- Member of SoldierX, BinRev, and Hack3r
- Twitter: @lattera

Disclaimers

- Opinions/views expressed here are mine, not my employer's
- Talk is semi-random
  - Tied together at the end
- Almost nothing new explained
  - Theory known
  - New technique
- Presentation and tools only for educational purposes

Assumptions

- Linux? What's that?
  - Concepts carry over to Windows and OSX
- Basic knowledge of C and 32bit Linux memory management
- Ability and desire to think abstractly
- Non-modified memory layout (NO grsec/pax)

History

- CGI/Web App vulnerabilities
  - Needed connect-back shellcode
  - Needed reliable, random access
    - Firewall holes are a problem
    - Needed way to reuse existing connection to web server
  - Needed to covertly sniff traffic
  - Libhijack is born (discussed later)

Setting the Stage

- Got a shell via CGI/Web App exploit
  - Reliable way to get back in
  - Apache good candidate
    - Already listening for connections
  - Modify apache process somehow to run a shell when a special string is sent
    - i.e. GET /shell HTTP/1.1

    $ whoami

    apache
  - Need to hook certain functions in runtime

Current Techniques

- Store shellcode on the stack
  - Stack is non-executable
- Store shellcode at $eip
  - Mucks up original code
- Store shellcode on the heap
  - Heap is non-executable
- LD_PRELOAD?
  - Process has already started

Process Loading

- execve is called
- Kernel checks file existence, permissions, etc.
- Kernel loads RTLD (Runtime Linker (ld.elf.so))
- Kernel loads process meta-data, initializes stack
  - Meta-data loaded at 0x08048000 on Intel 32bit Linux

Runtime Linker

- Loads process into memory
- Loads dependencies (shared objects)
    - DT_NEEDED entries in the .dynamic section
    - Patches PLT/GOT for needed dynamic functions
- Calls initialization routines
- Finally calls main()

ELF

- Executable and Linkable Format
- PE-COFF based on ELF
- Meta-data
- Tells RTLD what to load and how to load it

ELF

- Describes where to load different parts of the object file
  - Process Header (PHDR) – Minimum one entry; contains virtual address locations, access rights (read, write, execute), alignment
  - Section Header (SHDR) – Minimum zero entries; describes the PHDRs; contains string table, debugging entries (if any), compiler comments
  - Dynamic Headers – Contains relocation entries, stubs, PLT/GOT (jackpot)

Process Tracing

- Ptrace – Debugging facility for Linux
  - Kernel syscall
  - GDB relies on ptrace
  - Read/write from/to memory
  - Get/set registers
  - Debugee becomes child of debugger
  - Destructive
    - Original ptrace engineer evil, likely knew it could be abused

Allocating Memory

- We have arbitrary code to store. Where?
- Allocate memory in child
  - Unlike Windows and OSX, we cannot allocate from the parent process, the child must allocate
- Find “int 0x80” opcode
- Program's main code won't call kernel
  - Calls library functions which call the kernel
    - Libc!
  - Find a library function that calls the kernel by crawling the ELF meta-data

Allocating Memory - Finding “int 0x80”

- Loop through the ELF headers
  - Main ELF header contains pointer to PHDR
  - PHDR contains a pointer to the Dynamic headers
  - Dynamic headers has a pointer to the GOT
  - GOT[1] contains a pointer to the linkmap
  - linkmap is a structure created/maintained by RTLD and dlopen
  - linkmap points to each shared object’s ELF headers
  - Loop through symbol table of each shared object

Allocating Memory

- Parse ELF headers, loaded at 0x08048000
  - Headers include lists of loaded functions
- Found “int 0x80” in a shared object
- Back up registers
- Set $eip to address of found “int 0x80” opcode
- Set up stack to call mmap syscall
- Continue execution until mmap finishes

Injecting Shellcode

- After calling mmap
  - $eax contains address of newly-allocated mapping
  - Can write to it
    - Even if mapping is marked non-writable (PROT_READ | PROT_EXECUTE)
  - Restore the backed-up registers
  - Push return address
    - Shellcode needs to know where to return to
    - Decrement $esp by sizeof(unsigned long)
    - Copy $eip to $esp

Injecting Shellcode

- Write shellcode to newly-allocated mapping
- Set $eip to address of the shellcode
- Detach from the process
- Sit back, relax, and enjoy life
- But wait! There's more!

Hijacking Functions

- Global Offset Table/Procedure Linkage Table
  - Array of function addresses
- All referenced functions are in GOT/PLT
- PLT/GOT redirection
  - Shellcode["\x11\x11\x11\x11"] = @Function
  - GOT[@Function] = @Shellcode

Hijacking Functions

- Be careful
  - Multiple shared objects implement functions of the same name
    - Different signature
  - Make sure you target the correct function
  - Know your target
  - Set up a VM, mimicking the victim
    - Same OS, same patch levels, etc.
- Cannot reliably remove hijack

Injecting Shared Objects

- Why?
  - Don't have to write a ton of shellcode
  - Write in C, use other libraries, possibilities are endless
- Two ways of doing it
  - The cheating way: Use a stub shellcode that calls dlopen()
  - The real way: rewrite dlopen()

The Cheating Way

- Allocate a new memory mapping
- Store auxiliary data in mapping
  - .so path
  - Name of the function to hijack
  - Stub shellcode
- Stub shellcode will:
  - Call dlopen and dlsym
  - Replace GOT entry with entry found via dlsym

The Cheating Way

- Advantages
  - Easy
  - Extendable
  - Fast
- Disadvantages
  - Entry in /proc/pid/maps
  - Rely on stub shellcode

The Real Way

- Reimplement dlopen
  - Load dependencies (deps can be loaded via real dlopen)
  - Create memory maps
  - Write .so data to new memory maps
  - Patch into the RTLD
  - Run init routines
  - Hijack GOT

The Real Way

- Advantages
  - Completely anonymous
  - Extensible
- Disadvantages
  - Takes time to research and implement

Shared Objects

- Shared objects can have dependencies
- Shared objects have own PLT/GOT
  - Loop through Dynamic structures found in linkmap
  - Use same PLT/GOT redirection technique against shared objects
  - Even shared objects loaded via dlopen

Libhijack

- Libhijack makes injection of arbitrary code and hijack of dynamically-loaded functions easy
  - Shared objects via the cheating method
  - Inject shellcode in as little as eight lines of C code
  - Full 32bit and 64bit support
  - Support for FreeBSD/amd64
  - Interest in porting to OSX
- Always looking for help
- https://github.com/lattera/libhijack

Libhijack Release 0.6

- Version 0.6 released last night
  - Port to FreeBSD/amd64
  - Bug fixes

Libhijack TODO

- Version 0.7
  - Inject shared objects via “The Real Way”
  - Looking for an adventure? Port to Android
- Always looking for help

Prevention

- Make sure PLT/GOT entries point to correct lib
  - How? Symbol table resolution?
- Use dtrace, disable ptrace
  - From Solaris
  - Non-destructive debugging
- Limit ptrace usage (apache user shouldn't use it)

Prevention

- Static binaries
    - Major disk and memory usage
- Hypervisor?
- Grsec/PaX
    - Only protects to a certain extent
- Static and dynamic profiling
    - Watch for changes in GOT
    - Make sure changes reflect static profile
    - What about shared objects loaded via dlopen()?

FreeBSD

- FreeBSD's runtime linker
  - Much different than GNU's
    - Much easier on the eyes
  - link_map isn't as big of a deal
  - struct Struct_Obj_Entry
    - RTLD hacker's wet dream
    - Contains every single calculation libhijack needs
    - Located at GOT[1] -- same as GNU's link_map
    - Likely going to make hooking manually-injected .so's difficult

Demo

Assembly loading .so

exit(0);

Comments/questions
Thanks