One Shellcode To Rule Them All.

Who are we

- Michael “Borski” Borohovski
  - Co-Founder / CTO @ Tinfoil Security
  - Member of “Samurai” CTF team
  - MIT Computer Science
  - Hacking since 13, won Defcon 20 CTF

- Shane “ShaneWilton” Wilton
  - Engineer @ Tinfoil Security
  - Member of “Samurai” CTF team
  - University of Waterloo Computer Science
  - Hacking since he was just a wee little baby

TINFOIL SECURITY

Who are we

- Best web application scanner on the market
- Focused on Dev. and DevOps integrations
  - Empower developers to find and fix vulnerabilities before they're deployed
  - Enable security teams to focus on the hard problems
- Email secuinside@tinfoilsecurity.com for 2 free months

TINFOIL SECURITY

What is shellcode?

- Small piece of (assembled) code used as payload to exploit a vulnerability
- Common goals
  - Launch a shell
  - Read a file
  - Stage a larger payload
  - ???
- Lots of public examples of shellcode
  - Shell-storm, metasploit, etc.

TINFOIL SECURITY

Why write your own?

- Often you need to perform unique actions.
  - Unlock a door, call out to a different binary, etc.

- May have unique constraints
  - Can’t contain the $ character
  - Only alphanumeric characters
  - Runs under both little and big endian (Dalvik?)

- Fun!

TINFOIL SECURITY

How do you write shellcode?

- Learn the system calls for your platform
  - Man pages are your friend
- Start simple, then build more complexity
  - First, just call the _exit syscall
  - Then, “hello world” followed by _write followed by _exit
  - …
- Familiarize yourself with different calling conventions
  - x86 - cdecl, fastcall, etc.
  - PowerPC - registers
  - SPARC - register windows
    - Have fun with this one, because we didn’t
- Most importantly…

TINFOIL SECURITY

How do you write shellcode?

- Comment
- Comment
- Comment
- Commenting with ; is your best friend
- If you think understanding your Ruby code a month later is tough, try deciphering shellcode you've optimized to fit into a tiny buffer

TINFOIL SECURITY

Why is multiplatform shellcode useful?

- Deploy once, pwn always
- Consider the recent futex bug
  - Allowed for priv. esc. on linux
  - The original proof of concept (PoC) was for x86
  - Geohot used the bug to root an android phone (ARM)
  - Theoretically, a multi-platform payload could root **any** linux device
- Difficult to probe architectures in the wild
  - Same version of software can run on completely different architectures
    - Common with routers, smart devices, etc.
  - Guess wrong, and the target crashes
    - Crashes lead to detection

TINFOIL SECURITY

Why is multiplatform shellcode useful?

- Malware (but that's bad, don't do it!)
- Internet of things – everything connected, built differently, lots of cheap hardware choices
- "100,000 Refrigerators and other home appliances hacked to perform cyber attack"
- Internet census 2012 attacked 1.2M devices
    - Exploit/binary targeted **9** different platforms/architectures.

TINFOIL SECURITY

Compiling your pieces

- QEMU or Virtual Machine (VMWare, Parallels, etc.)
- Write shellcode once
  - Load image for desired architecture in QEMU
    - `qemu-img create -f qcow2 linuxppc.qcow2 5G`
    - ```
      qemu-system-ppc -hda linuxppc.qcow2 \
                      -cdrom debian-ppc.iso \
                      -boot d \
                      -m 512
      ```
  - Use nasm to assemble once in qemu
    - `nasm -f bin shellcode.asm`

TINFOIL SECURITY
      ```

Compiling your pieces

- Capstone
  - Programmable disassembly framework
  - http://www.capstone-engine.org/
  - *Arm*, *Arm64* (*Armv8*), *Mips*, *PowerPC*, *SPARC*, *SystemZ*, *XCore* & *Intel*
  - Written in C but bindings for Ruby, Python, etc.

- Useful for seeing how opcodes disassemble in different architectures
  - Same opcode under different architectures lead to different behaviors
  - Take shellcode, print out disassembly for ARM, PPC, X86, etc.

TINFOIL SECURITY

Multi-Platform Payloads

- Different architectures require different payloads
- Each architecture has its own nuances
  - x86 has variable length instructions
  - *SPARC* has fixed-length 32-bit instructions
  - Shellcode must not crash on any platform
- We have three goals
  - Write payloads for each architecture
  - Determine the architecture of the CPU
  - Jump to the payload for that architecture
- How do we determine the architecture of the CPU?

TINFOIL SECURITY

CPU Switch Header

- The same bytes decode to different instructions on different architectures
- A jump instruction on x86 might be a NOP on PowerPC
- Example - “\x37\x37\xeb\x78”
  - x86
    - aaa; aaa; jmp 116+4
  - MIPS
    - ori $s7, $t9, 0xeb78
  - *SPARC*
    - sethi %hi(0xdfade000), %i3

TINFOIL SECURITY

Finding “switch” instructions

- Needs to jump in one architecture, and be NOP-like in all others
  - Can’t crash any architectures
  - Can’t modify PC
  - We don’t care about most other register state
- Most architectures encode branch instructions in predictable formats.
  - SPARC - 00-a-bbbbb-010-<22-bit offset>
  - a - 1-bit annulment flag
  - bbbbb - 5-bit condition
- We can fuzz all of the possible branch instructions!

TINFOIL SECURITY

Choosing Jump Candidates

- Compute all of the branch instructions for an architecture
- Use Capstone to decode them in all other targeted architectures
- Look for instructions which decode harmlessly in most other architectures
- Easier than it sounds!
  - We structure our switch-table like an onion
  - “Peel” off an architecture with each instruction
  - i.e. an instruction can’t crash MIPS, if MIPS has already jumped to its payload by that point

TINFOIL SECURITY

Dependency Resolution

- Consider the case on two architectures, **A** and **B**
- Let $\mathbf{I_A}$ and $\mathbf{I_B}$ be the sets of possible branch instructions for **A** and **B**
  - $\mathbf{I_A} = \{a_1, a_2, a_3\}$
  - $\mathbf{I_B} = \{b_1, b_2, b_3\}$
- Let $\mathbf{D_{in}}$ be the set of dependencies for instruction **in**
  - i.e. if $\mathbf{a_1}$ crashes on architecture **B** then $\mathbf{D_{a1}} = \mathbf{I_B}$
- We need an instruction from $\mathbf{I_A}$ and an instruction from $\mathbf{I_B}$ such that there exists an evaluation order which resolves all dependencies
  - Called a topological ordering on the dependency graph

TINFOIL SECURITY

Algorithmically...

1. Let **S** be the cartesian product of the sets of branch instructions
2. For each **s** $\in$ **S** = (a, b, c, ...)
    a. Construct a graph **G** with vertex set given by the elements of **s**
    b. Create a directed edge from vertex **i** to vertex **j** if instruction **i** crashes under the architecture for which **j** originates from
    c. Check for a topological ordering on **G**
        i. If one exists, return it, we are done
        ii. Otherwise, continue
3. If no ordering exists, we need to be clever
    a. Consider multi-stage payloads which split the targeted architectures into more manageable groupings

TINFOIL SECURITY

Putting It Together

- Polyglot @ DEFCON 22 CTF Quals
  - Construct a payload which reads a flag on x86, ARMEL (little endian), ARMEB (big endian), and PPC

- The dependencies are resolvable as:
  - x86 -> PPC -> ARMEB -> ARMEL
  - tsort can do most of this work from the command-line!

	73 12 00 00	48 00 01 70	9A 00 00 40	13 00 00 EA
x86	jae 0x14	-	-	-
PPC	andi r1, r0, 72	bdnzfa- lt, 0x98	-	-
ARMEB	tstvc r2, #0	stmdami r0, ...	bls 0x110	-
ARMEL	...	...	...	b 0x60

TINFOIL SECURITY

Putting It Together Cont.

- Each architecture is jumping to a different point, so we can simply insert our platform-specific shellcode at the correct offsets
- Note the strange instructions
  - 48 00 01 70 -> bdnzfa- lt, 0x98
  - Not a terribly useful instruction, but acts like a simple branch in our case
- You just owned four different platforms with one payload
  - Congratulations!

TINFOIL SECURITY

To sum it all up

- Hardware is becoming more and more varied, and will only get further fragmented over time
- Knowing and being able to fingerprint one architecture will become a thing of the past
- Writing one payload that works across many architectures was once a luxury, but is quickly becoming a requirement for launching attacks in the wild

TINFOIL SECURITY

To sum it all up

- Basic idea: set up a jump table at the beginning of your shellcode, with one architecture falling through with each instruction
- Find jmp/branch instructions in one architecture that are NOPs or NOP-like instructions in all others you're targeting
- To automate this search, you can reduce the problem to one of dependency resolution

TINFOIL SECURITY

감사합니다

TINFOIL SECURITY