Using Symbolic Execution for IoT Bug Hunting

X-Force Red

Our Mission: Hacking Anything to Secure Everything

Presenters:
Grzegorz Wypych, X-Force Red

Bio

Age - 36

Full name - Grzegorz Wypych (h0rac)

- Career path: Network Engineer => Network Architect => Software Developer => Security Researcher
- Languages: C, python, node.js, javascript, Java
- Papers: Yeah CCIE R&S - will expire in 25/06/2019 together with other Cisco certs ;]
- Overall 15 years IT experience
- ARM/MIPS assembly enthusiast :)
- 0day CVEs on account related to TP-LINK devices
- When I do no research: I build fishing rods and fish out of the water :)
- I wish my day to have more than 24 hours :)
- Motto ? Before use.. disassemble :)

https://twitter.com/horac341

https://github.com/h0rac/

1) Problems with traditional dynamic vulnerability research

- You test where you are and not where you want to be
- Anti-debuggers applied
- Busybox without tftp, can't upload gdbserver, or core dump not available
- No ssh, no shell, no access (JTAG, UART)
- Qemu emulation nightmare

You test where you are and not where you want to be

```
0x00408138 in http_parser_main ()
gef➤  next
[ Legend: Modified register | Code | Heap | Stack | String ]

$zero: 0x0
$at  : 0x80353a0c
$v0  : 0x0
$v1  : 0x74
$a0  : 0x7f81b7f6  →  0x00000a00
$a1  : 0x00413a62  →  0x4b4f0000
$a2  : 0x0
$a3  : 0x0
$t0  : 0x4
$t1  : 0x30
$t2  : 0x64
$t3  : 0x7f81b439  →  0x00000031 ("1"?)
$t4  : 0x7f81b430  →  "172.16.0.1"
$t5  : 0x4
$t6  : 0x2e
$t7  : 0x0
$s0  : 0xffffffff
$s1  : 0x0042dd84  →  0x0042d198  →  0x0042dd84  →  [loop detected]
$s2  : 0x7f81b7e0  →  "http://tplinkwifi.net"
$s3  : 0x00410000  →  0x00002021 ("!"?)
$s4  : 0x00430000  →  0x00000000
$s5  : 0x00413a24  →  "JSESSIONID"
$s6  : 0x00413c10  →  0x00408330  →  <http_parser_main+3568> move s0, zero
$s7  : 0x0042d8a4  →  0x00413b98  →  "Authorization"
$t8  : 0x1
$t9  : 0x76f40860  →  <strncmp+0> sltiu v0, a2, 4
$k0  : 0x20
$k1  : 0x0
$s8  : 0x7f81b7e8  →  "tplinkwifi.net"
$pc  : 0x00408140  →  <http_parser_main+3072> lw gp, 16(sp)
$sp  : 0x7f81b4f0  →  0x00000000
$hi  : 0x1
$lo  : 0x0
$fir : 0x0
$ra  : 0x00408140  →  <http_parser_main+3072> lw gp, 16(sp)
$gp  : 0x00435630  →  0x00000000

0x7f81b4f0│+0x0000: 0x00000000   ← $sp
0x7f81b4f4│+0x0004: 0x00000000
0x7f81b4f8│+0x0008: 0x00000000
0x7f81b4fc│+0x000c: 0x00000000
0x7f81b500│+0x0010: 0x00435630  →  0x00000000
0x7f81b504│+0x0014: 0x00000000
0x7f81b508│+0x0018: 0x00000000
0x7f81b50c│+0x001c: 0x00000000

     0x408134 <http_parser_main+3060> addiu  a1, a1, 14932
     0x408138 <http_parser_main+3064> jalr   t9
     0x40813c <http_parser_main+3068> li     a2, 14
 →   0x408140 <http_parser_main+3072> lw     gp, 16(sp)
     0x408144 <http_parser_main+3076> bnez   v0, 0x408178 <http_parser_main+3128>
     0x408148 <http_parser_main+3080> move   a0, s8
     0x40814c <http_parser_main+3084> lw     t9, -31724(gp)
     0x408150 <http_parser_main+3088> nop
     0x408154 <http_parser_main+3092> jalr   t9

[#0] Id 1, Name: "httpd", stopped, reason: SINGLE STEP

[#0] 0x408140 → http_parser_main()
[#1] 0x406840 → http_inetd_main()
[#2] 0x404268 → http_init_main()
[#3] 0x4034c0 → main()

0x00408140 in http_parser_main ()
gef➤
```

Traditional Debugging with GDB

- Hard to identify proper breakpoint place
- You follow single path
- Changing path selection with register modify could break execution
- Once debug fail, you need to start from the beginning.
- If this is remote debug session, you loose your breakpoints (Agh!!!)
- Time consuming !

But you can see step by step what is going on in process memory

Anti-debuggers applied

```
0042bcb8  int32_t (* const cmem_updateFirmwareBufFree@GOT)() = cmem_updateFirmwareBufFree
0042bcbc  int32_t (* const rdp_oidToOidStr@GOT)() = rdp_oidToOidStr
0042bcc0  int32_t (* const signal@GOT)() = signal
0042bcc4  int32_t (* const dm_compareNumStack@GOT)() = dm_compareNumStack
0042bcc8  int32_t (* const dm_validateString@GOT)() = dm_validateString
```

```
0042bd7c  int32_t (* const sscanf@GOT)() = sscanf
0042bd80  int32_t (* const sigaction@GOT)() = sigaction
0042bd84  int32_t (* const setsid@GOT)() = setsid
0042bd88  int32_t (* const g_oidStringTable@GOT)() = g_oidStringTable
```

When you try to hit breakpoint in debugger and step over or continue,
instead going to expected destination you land in **SIGTRAP** ;/

To avoid, you can:

a) Try to patch binary, but no guarantee If this will work
b) Try to use GDB for software debug bypass
c) Modify registers to not execute during dynamic debugging

And then: We lost debug session… UPS :(

But we are missing our goal ! We don't want to spend time on avoiding anti-debuggers
We want to utilize our time for vulnerability research and exploitation :)

Busy box without tftp, can’t upload gdbserver or core dump not available

Standard binaries available under IoT OS usually have binding to busy box. First step After image retrieve is to check available commands under busybox. If we are lucky enough And tftp/ftp is available we have option to upload gdbserver binary for dynamic analysis, however Sometimes busybox is intentionally limited - What we can do then ?

We can try to reflash image with own busybox, but no guarantee it will work

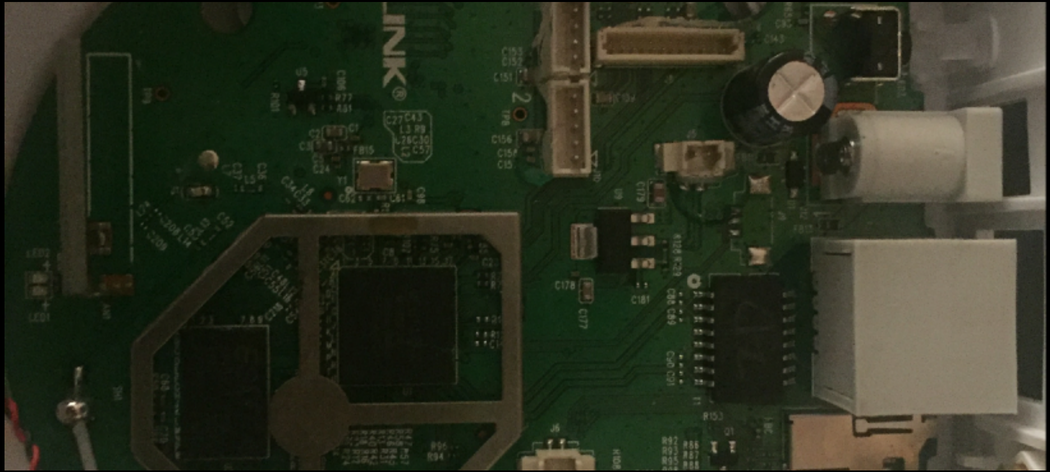
If we want to have core dump for analysis, usually this commands enable it on device

```
ulimit -c unlimited
echo /var/tmp/core > /proc/sys/kernel/core_pattern
```

In some cases this do not work and you cannot grab core dumps for analysis

No ssh, no shell, no access

- Sometimes ssh is available but not for us :) What usually happens (example on TP-LINK devices), ssh is available only for certain application like Tether mobile app which is used for remote management.
- Telnet is usually limited to “cli” binary which is loaded in runtime. Every time user log in to device via telnet, binary is loaded to memory and provides limited config options.
- UART/JTAG not available, like on this IP Camera NC450



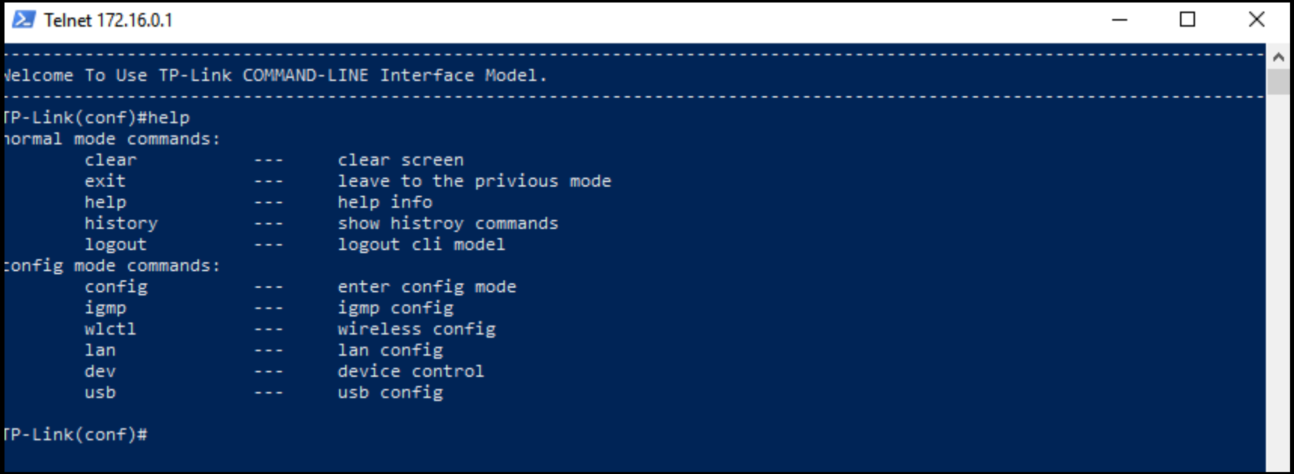
TP-LINK NC-450 board no JTAG/UART visible pins

Bonus - CLI binary TP-LINK devices

Just as research bonus :) - I found something interesting in “cli” binaries available in all TP-LINK devices I’ve researched. It has hidden menu with shell access, but to enable it, it is required To have active debug session and manipulate “flags" in memory:

```
set {int}0x41e9d8 = 0x14 – g_cli_user_level
set {int}0x0041e9dc = 0x14 – g_cli_mode
```

This is normal command-line tool available on TP-LINK devices



Bonus - CLI binary TP-LINK devices

This is how it looks like after modifying flags in memory

```
-------------------------------------------------------------------------------------------------------------------------------------------------------
Welcome To Use TP-Link COMMAND-LINE Interface Model.
-------------------------------------------------------------------------------------------------------------------------------------------------------
TP-Link(conf)#
TP-Link?help
normal mode commands:
        clear           ---     clear screen
        exit            ---     leave to the privious mode
        help            ---     help info
        history         ---     show histroy commands
        logout          ---     logout cli model
privilege mode commands:
        enable          ---     enter privilege mode
        sh              ---     force to cli
config mode commands:
        config          ---     enter config mode
        igmp            ---     igmp config
        wlctl           ---     wireless config
        lan             ---     lan config
        dev             ---     device control
        usb             ---     usb config

TP-Link?sh
[ doFshell ] cmd: sh
~ # ls -la
drwxr-xr-x   10       138 web
drwxr-xr-x   15         0 var
drwxr-xr-x    4        38 usr
dr-xr-xr-x   11         0 sys
drwxr-xr-x    2       276 sbin
dr-xr-xr-x   90         0 proc
drwxr-xr-x    2         3 mnt
lrwxrwxrwx    1        11 linuxrc -> bin/busybox
drwxr-xr-x    3      1138 lib
drwxr-xr-x    7       502 etc
drwxr-xr-x    8      1326 dev
drwxr-xr-x    2       388 bin
drwxr-xr-x   13       177 ..
drwxr-xr-x   13       177 .
~ #
```

I don’t know if they left dev code for debug purposes or smth but why it is under production code in every device ? :)

Qemu emulation nightmare

Everyone is saying Qemu can emulate IoT binaries/firmware, let's verify that against real software :)

- TP-LINK devices usually store in flash memory "shared region" where they store configuration options. During Qemu emulation we do not have access to and **strace** immediately inform us about that and fail emulation.

```
sudo chroot . ./qemu-mipsel-static -strace usr/bin/httpd
```

```
40411 ipc(23,1234,0,950) = -1 errno=22 (Invalid argument)
40411 write(1,0x7630f278,92)[ dm_shmInit ] 086:  shmget to exitst shared memory failed. Could not create shared memory.
 = 92
40411 ipc(1,-1,1,0) = -1 errno=22 (Invalid argument)
40411 write(1,0x7630f278,53)[ dm_acquireLock ] 252:  lock failed, errno=22 rc=-1
 = 53
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
(angr) → rootfs
(angr) → rootfs
(angr) → rootfs sudo chroot . ./qemu-mipsel-static -strace usr/bin/httpd
```

```
00404328  lw     $t9, -0x7d14($gp)  {dm_shmInit@GOT}
0040432c  nop
00404330  jalr   $t9
00404334  move   $a0, $zero  {0x0}
00404338  jal    sub_403fb8
```


X-Force Red

2) Write plugin for Binary Ninja

Before we jump to symbolic execution, let’s talk about Binary Ninja Disassembler

Pros:

- Nice python api
- Cheaper than IDA Pro
- Support multi-processors (ARM/MIPS/PowerPC etc)
- Modern UI :)
- Multi-disassembler options: Medium IL, ILL etc

Cons:

- Less features than IDA Pro
- Less processor types support
- No C decompiler

I think creators of Binary Ninja provides standard functionality to Disassembler, but leave a lot for users to add as plugins, and this is where power is unlimited

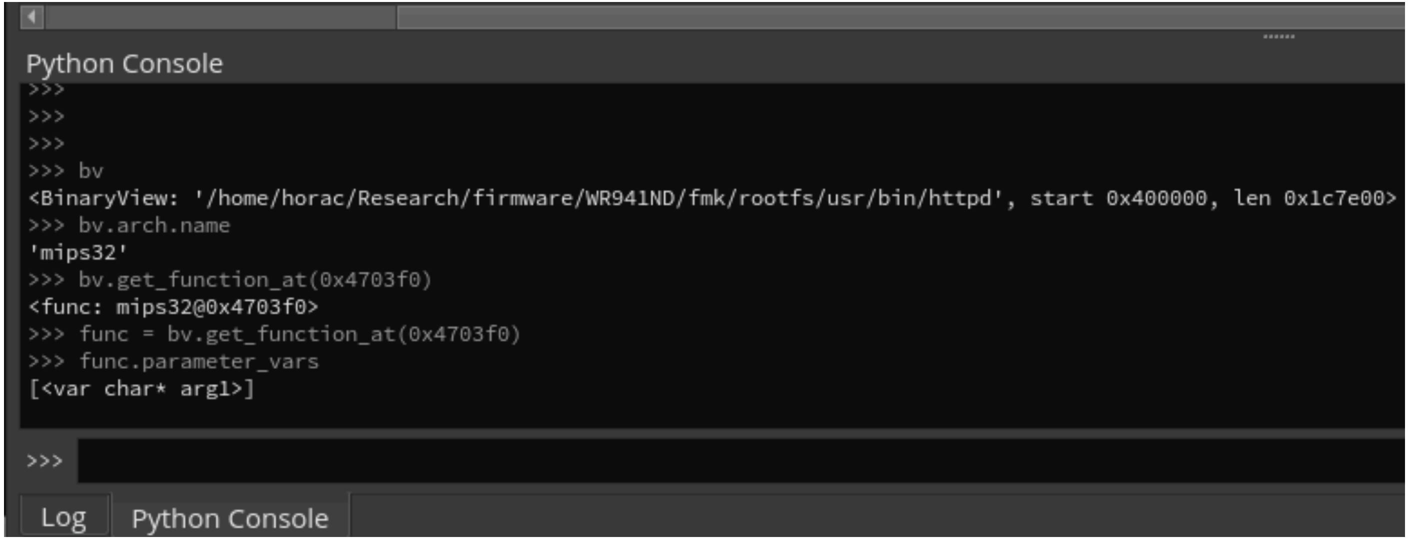
Important Binary Ninja components

- When binary is loaded **bv** reference is available for us
- BinaryView and Architecture class allows to take basic information from analyzed binary (architecture, endianness, functions and their params etc)
- Most common utilized modules:

A) plugin - provides core for UI (PluginCommand, BackgroundTaskThread)

B) interaction - provides different UI components

C) highlight - colors for graph view



Plugin module
(PluginCommand class)

There are two ways to use **PluginCommand** class from plugin module

Use direct PluginCommand class in main python file

```
PluginCommand.register(
    "Explorer\WR941ND\Explore", “Description”, BackgroundTaskManager.vuln_explore)
```

Encapsulate in separate class by inheritance

```
class UIPlugin(PluginCommand):

    def __init__(self):
        super(UIPlugin, self).register_for_address("Explorer\WR941ND\Start Address\Set",
        "Set execution starting point address", self.set_start_address)
        super(UIPlugin, self).register("Explorer\WR941ND\Start Address\Clear”,
```

Explanation on function and parameters:

`register` – expect handler with one param, **bv** instance
`register_for_address` – expect handler with two params, **bv** instance and **address**

“\” is important it allows to create sub-menus

Plugin module

(BackgroundThread class)

To execute actions in BinaryNinja, we need to inherit from **BackgroundTaskThread**
And override **run** method by our implementation

```
class AngrRunner(BackgroundTaskThread):
    def __init__(self, bv, explorer):
        BackgroundTaskThread.__init__(
            self, "Vulnerability research with angr started...", can_cancel=True)
        self.bv = bv
        self.explorer = explorer

    def run(self):
        self.explorer.run()
```

We can define own parameters for __init__ constructor. Here we provide own explorer instance which in this example could be VulnerabilityExplorer, ROPExplorer, JSONExploitCreator, FileExploitCreator

Interaction module

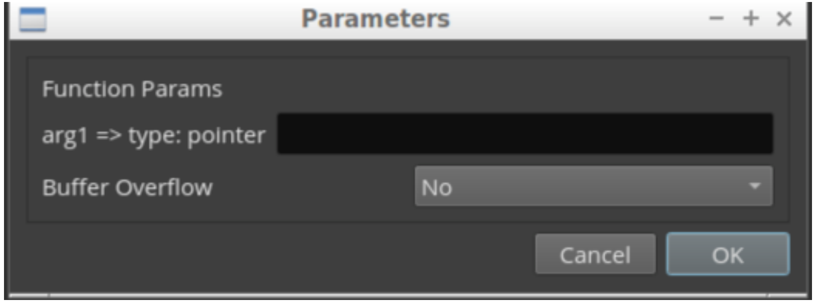
UI components are provided by **interaction** module. They are very easy to use

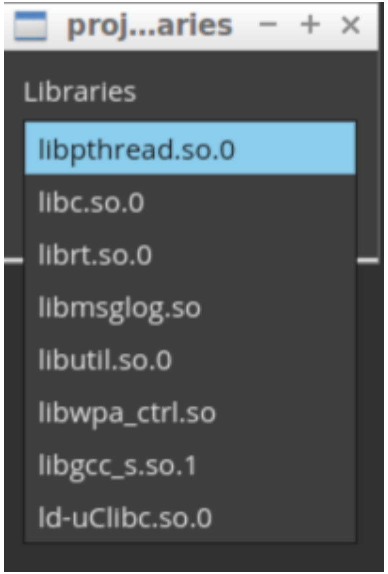
```
def generate_menu_text_fields(self, arg_types):
       menu = ["Function Params"]
       for arg in arg_types:
           text_field = interaction.TextLineField("{0} =>
                                   type: {1}".format(arg['param'], arg['type']))
           overflow_field = interaction.ChoiceField(
               "Buffer Overflow", ["No", "Yes"])
           menu.append(text_field)
           menu.append(overflow_field)
       return menu

 menu_items = self.generate_menu_text_fields(mapped_types)
       menu = interaction.get_form_input(menu_items, "Parameters")
```

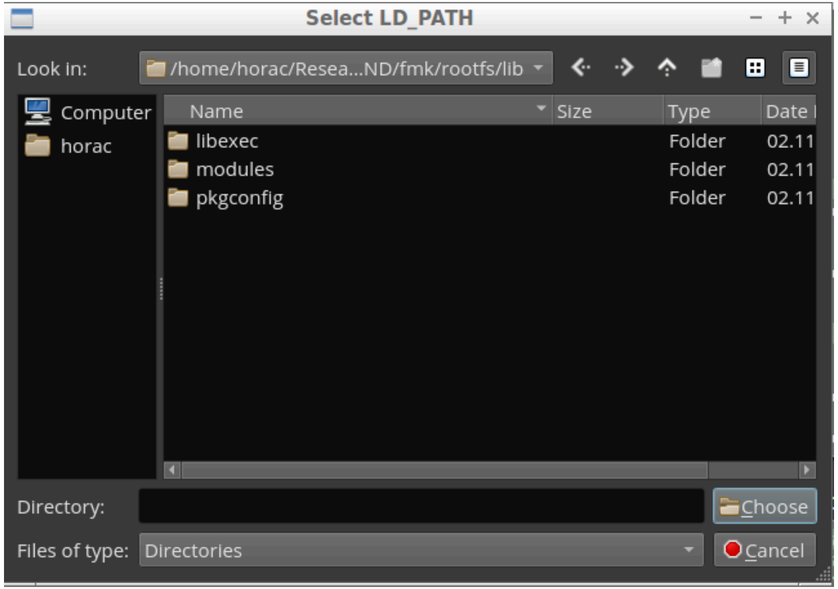
We can create UI components separately or use function **get_form_input** to create our custom menu. Function expect list of fields we want to include (TextLineField, ChoiceField etc). It returns also list of results

Interaction module (sample UI)

Sample UI look of components.







Highlight module

```
@classmethod
 def color_path(self, bv, addr):
     # Highlight the instruction in green
     blocks = bv.get_basic_blocks_at(addr)
     UIPlugin.path.append(addr)
     for block in blocks:
         block.set_auto_highlight(HighlightColor(
             HighlightStandardColor.GreenHighlightColor, alpha=128))
         block.function.set_auto_instr_highlight(
             addr, HighlightStandardColor.GreenHighlightColor)
```

This is example function used for path coloring during symbolic execution. We first get basic blocks Of assembly by address and highlight them to whatever color we want. Later also single addresses are colored. Results are store in class variable path.

We can call this function from any place, but in plugin I will present I use it during symbolic execution

3) How you can search for vulnerabilities without hacking physical device access or without Qemu emulation

- angr features we will use
- We will look on CVE -2019-6989 Buffer Overflow WR941ND (MIPS)
- We will identify vulnerable code with basic static analysis
- We will confirm vulnerability with symbolic execution (well.. tuned a little :)) using created plugin

And guess how ? We will not even try to run firmware, we will emulate it with **angr**

angr features

Load/Save to emulated memory

```
state.memory.store(sp+0x2c, state.solver.BVV(self.gadget3, 32))
state.memory.load(0x100, size)
```

Load/Save to register

```
state.regs.s0 = 0x100
state.regs.s1 = "AAAA"
pc = state.solver.eval(state.regs.pc, cast_to=int)
s1 = state.solver.eval(state.regs.s1, cast_to=bytes)
```

CFG Analysis

```
self.proj.analyses.CFGFast(regions=[(self.func_start_addr, self.func_end_addr)])
```

Hooking

```
self.proj.hook(self.func_end_addr, self.overwrite_ra)
```

PointerWrapper

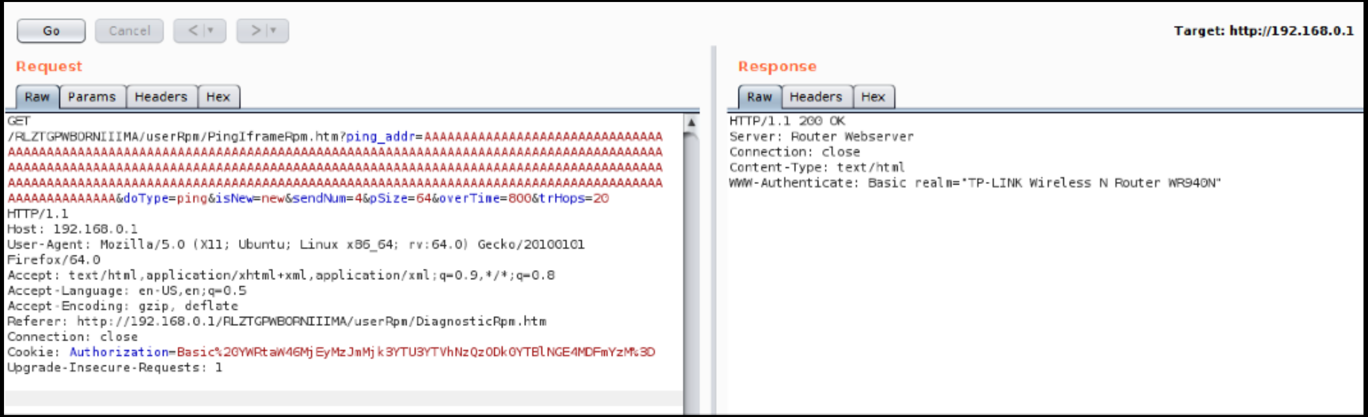
```
angr.PointerWrapper(item.get('value'))
```

Call state

```
self.proj.factory.call_state(self.func_start_addr, args['arg0'])
```

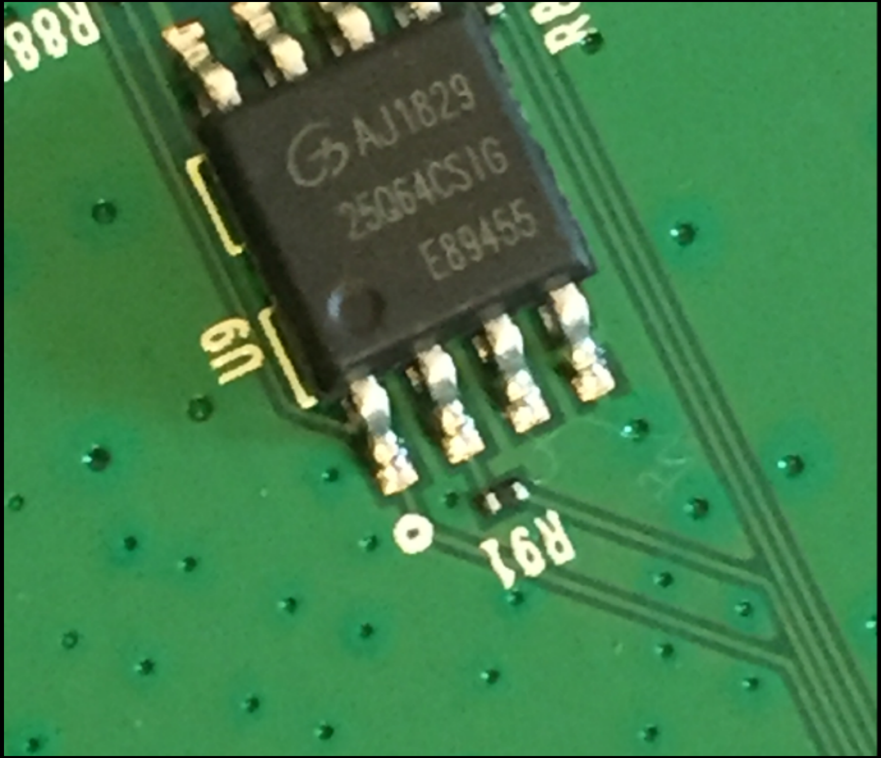
Find vulnerable Endpoint- W941ND

In Management panel, we have option to send health pings and check availability. However modyfing **ping_addr** with custom string crash httpd service in router. Now we know something is wrong but what exactly ???



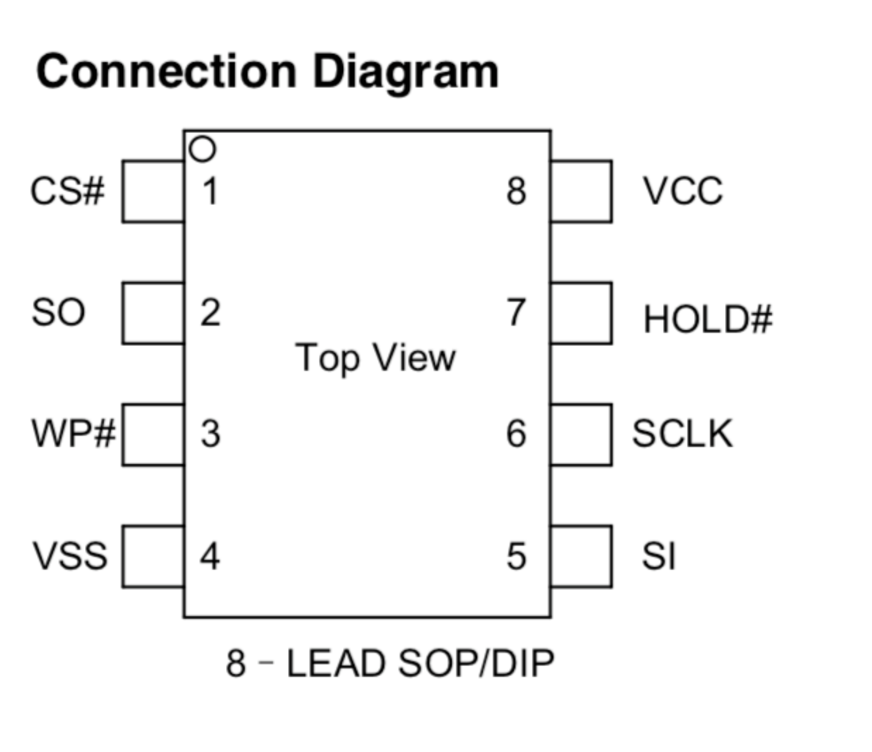
Let's dump firmware and start some basic static analysis and search for strings like URL endpoint or parameters names

Firmware dump -when device is available



Flash chip (GD25Q64C) on most TP-LINK devices - Archer C5 v4 - another device but same process

Firmware dump -when device is available

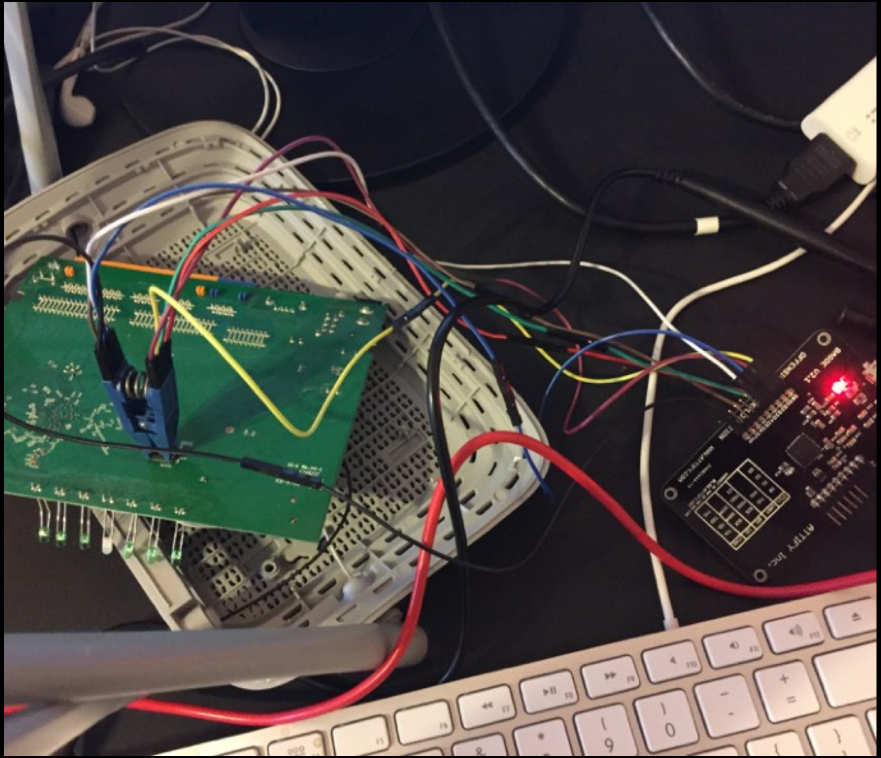


Chip info how to connect PINS for SPI

Firmware dump -when device is available

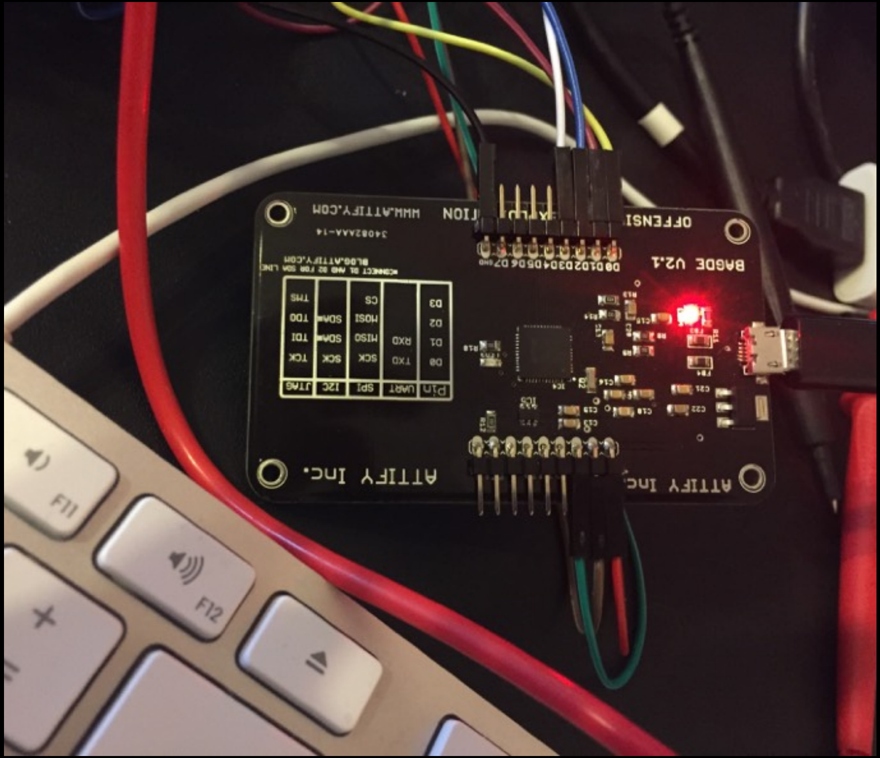
SOC8 clips connected to flash chip

Firmware dump -when device is available



Clips connected to flash chip and Attify Badge, bus pirate and any other SPI supported device will also work

Firmware dump -when device is available



Connected Attify badge over SPI

```
sudo ./flashrom -p ft2232_spi:type=232H -r firmware.bin
```

No device, but firmware available on vendor site

https://www.tp-link.com/us/support/download/tl-wr941nd/#Firmware

TL-WR941ND(US)_V6_151203		
Published Date: 2016-12-03	Language: English	File Size: 3.21 MB
Notes: TL-WR940N(US)3.0 / TL-WR941ND(US)6.0		

Next step is to extract firmware using firmware-mod-kit (easiest way) and find binaries in rootfs we want to analyse.

```
~/Research/firmware-mod-kit/extract-firmware.sh wr941nd.bin
```

Firmware - first look

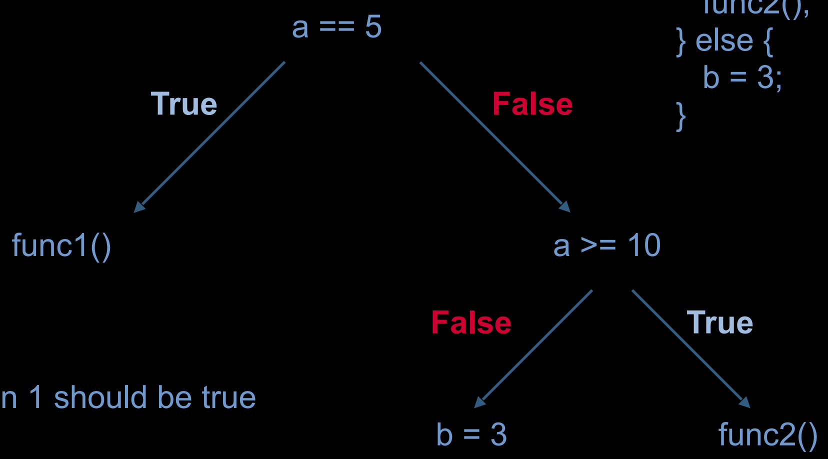
```
→  plugins cd ~/Research/firmware/WR941ND
→  WR941ND ls
fmk
→  WR941ND cd fmk/rootfs
→  rootfs ls
bin  dev  etc  lib  linuxrc  mnt  proc  qemu-mips-static  root  sbin  sys  tmp  usr  var  web
→  rootfs cd usr/bin
→  bin ls
[  arping  dbclient  dropbear  dropbearconvert  dropbearkey  httpd  lld2d  logger  scp  test  tftp
→  bin
→  bin
→  bin pwd
/home/horac/Research/firmware/WR941ND/fmk/rootfs/usr/bin
→  bin readelf -h httpd
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x41c5b0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x70001007, noreorder, pic, cpic, o32, mips32r2
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
→  bin readelf -d httpd

Dynamic section at offset 0x180 contains 28 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libpthread.so.0]
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
 0x00000001 (NEEDED)                     Shared library: [librt.so.0]
 0x00000001 (NEEDED)                     Shared library: [libmsglog.so]
 0x00000001 (NEEDED)                     Shared library: [libutil.so.0]
 0x00000001 (NEEDED)                     Shared library: [libwpa_ctrl.so]
 0x00000001 (NEEDED)                     Shared library: [libgcc_s.so.1]
 0x0000000c (INIT)                       0x41c524
 0x0000000d (FINI)                       0x543f30
```

Symbolic execution in nutshell

Example of symbolic execution tree base on ARM CPU instructions

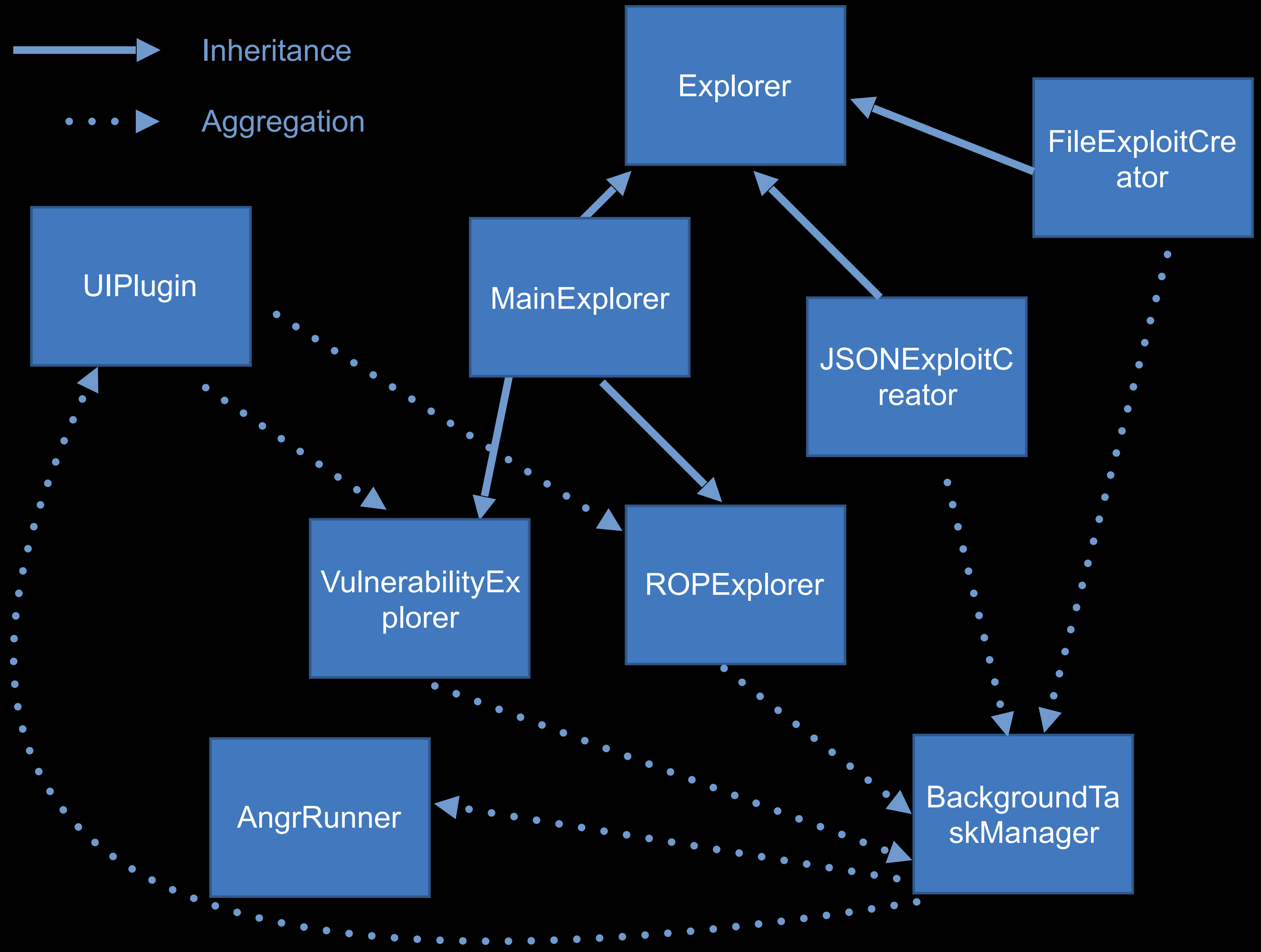
```
MOVS    R1, #0
LDR     R2, [R3]
CMP     R2, #5
BEQ     0x080002D2
LDR     R2, [R4]
CMP     R2, #10
BGE     0x8000258
MOVS    R1, #3
```

```
Int b = 0;
if (a == 5 ) {
   func1();
} else if (a >= 10) {
  func2();
} else {
  b = 3;
}
```



Possible values for a if condition 1 should be true
a = 5
Possible values for “a” if condition 2 should be true
a = 10,11,12,13….10000, 703933…
Possible values for “a” if condition 3 should be true
a = 6,7,8,9

Explorer view



Vulnerability exploration DEMO

Vulnerability exploration DEMO TIME :)

4) I have RA in control, let' s build PoC exploit without debugger /crash dump/memory snapshot ?

- MIPS Assembly in nutshell
- We will present features of angr we will use for ROP exploitation
- We will present gadgets for ROP
- We will create ROP chain
- We will execute ROP chain
- We will provide report for CPU registers and stack during ROP execution

And guess how ? We will not even try to run firmware, we will emulate it with **angr**

MIPS Assembly in nutshell

- Endianness: Little Endian(MIPSEL) and Big Endian(MIPS)
- First four arguments to function passed in registers ($a0, $a1, $a2, $a3)
- Function need more arguments ? They are pushed on stack
- Instruction Pointer aka intel EIP => $pc (Program counter)
- Stack pointer: $sp
- Calling function executed by loading register to $t9 and jalr $t9
- Return address stored in $ra
- Return value $v0
- Callee responsible to store value of registers before executing
- Space for local variables in stack frame: $sp, sp,- 0x3c in prologue

ROP gadgets

Gadget 1

```
0x00055c60:
        addiu $a0, $zero, 1; # prepare param for sleep func
        move $t9, $s1; # copy gadget2 address
        jalr $t9;
```

Gadget 2

```
0x00024ecc:
        lw $ra, 0x2c($sp); # load gadget 3 address
        lw $s1, 0x28($sp); # load sleep func addr
        lw $s0, 0x24($sp);  # load junk
        jr $ra;
```

Gadget 3

```
0x0001e20c:
        move $t9, $s1;
        lw $ra, 0x24($sp); # load gadget 4 address
        lw $s2, 0x20($sp); # load junk
        lw $s1, 0x1c($sp); # load gadget 5 address
        lw $s0, 0x18($sp); # load junk
        jr $t9;
```

ROP gadgets

Gadget 4

```
0x000195f4:
        addiu $s0, $sp, 0x24; # store in $s0 address of shell code
        move $a0, $s0; # copy shell code address to $a0
        move $t9, $s1; # copy address of gadget5 to $t9
        jalr $t9; # jump
```

Gadget 5

```
0x000154d8:
        move $t9, $s0; # copy address of $s0 to $t9
        jalr $t9; # execute shell code
```

Sleep function

```
0x00053ca0
```

ROP exploitation DEMO

ROP exploitation DEMO TIME :)

X-Force Red

THANK YOU

FOLLOW US ON:

- ibm.com/security
- securityintelligence.com
- xforce.ibmcloud.com
- @ibmsecurity
- youtube/user/ibmsecuritysolutions

IBM