Finding format string vulns with (and in) binary ninja

whoami

Vasco Franco (aka **jofra**)

- Lisbon, Portugal 🇵🇹
- CTF player @ **STT**
- Bug bounties from time to time

@V_jofra

Why am I here

- Wrote an article for **Paged Out!**
- Was chosen by gynvael and Rodrigo as the **best security/RE article**

Motivation

- Found a format string vuln while **fuzzing**
- Wanted to **look for similar vulnerabilities** in the binary (no source code)
- Decided to **model it using binary ninja**
- Found 1 similar vulnerability

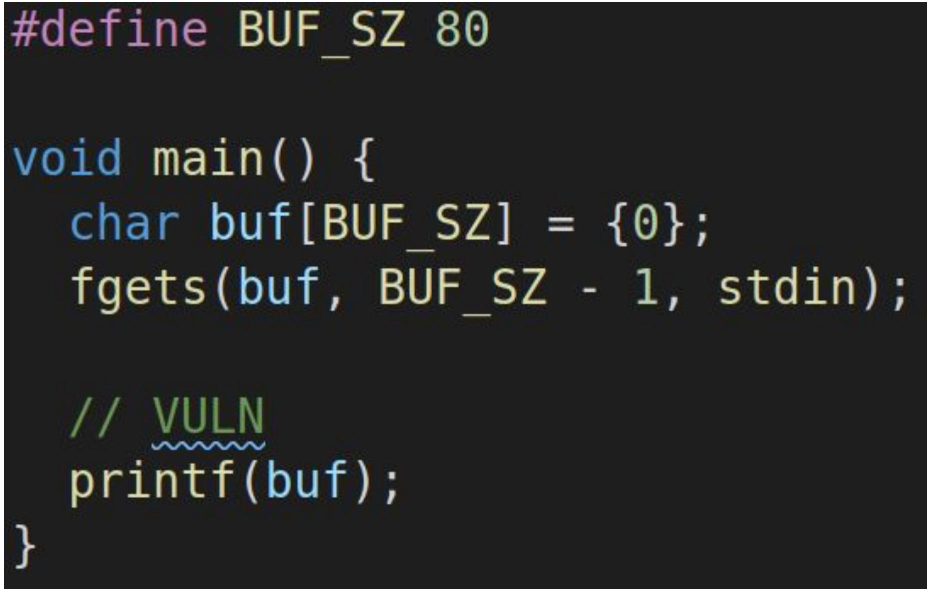
Agenda

1. Format string vulnerabilities
2. Modeling format string vulns
3. Binary Ninja
4. Using binja to model format string vulns
5. Results

Format string vulnerabilities

Format string vulnerabilities

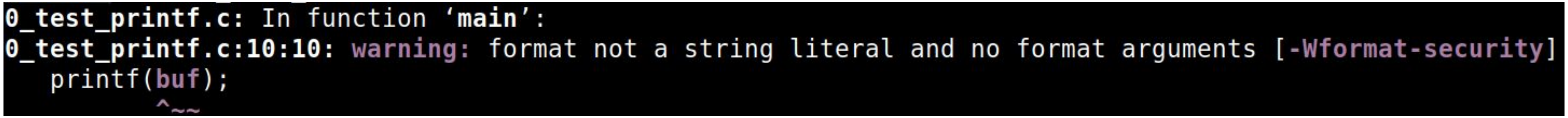
- Occurs when the **format argument** of a function from the printf family is **controlled by user input**
- printf(buf)
- printf(“%s”, buf)



```c
#define BUF_SZ 80

void main() {
  char buf[BUF_SZ] = {0};
  fgets(buf, BUF_SZ - 1, stdin);

  // VULN
  printf(buf);
}
```

Format string vulnerabilities

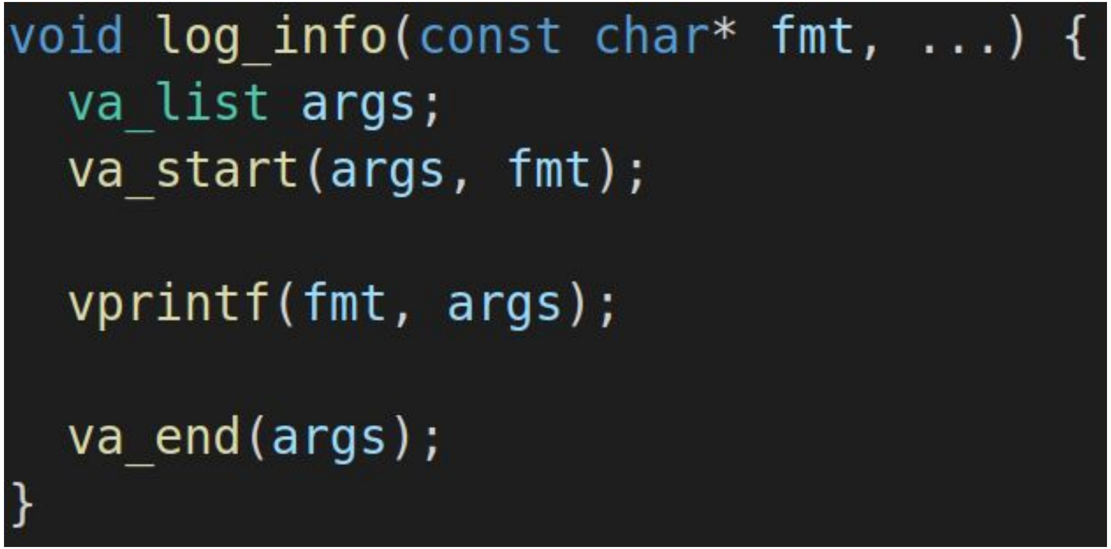
- Old and well researched vulnerability
- Compilers emit **warnings**



```
0_test_printf.c: In function ‘main’:
0_test_printf.c:10:10: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(buf);
          ^~~
```

Format string vulnerabilities

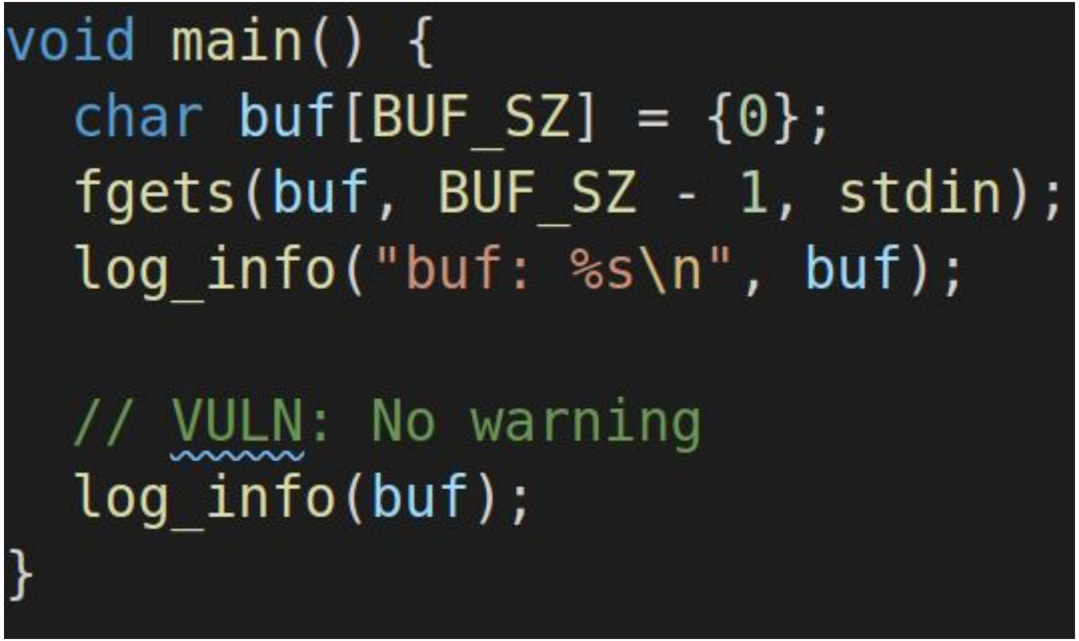
- We can write our own **custom printf wrappers** (common for logging functions)
- We will call these **printf-like functions** from now on



```
void log_info(const char* fmt, ...) {
  va_list args;
  va_start(args, fmt);

  vprintf(fmt, args);

  va_end(args);
}
```

Format string vulnerabilities

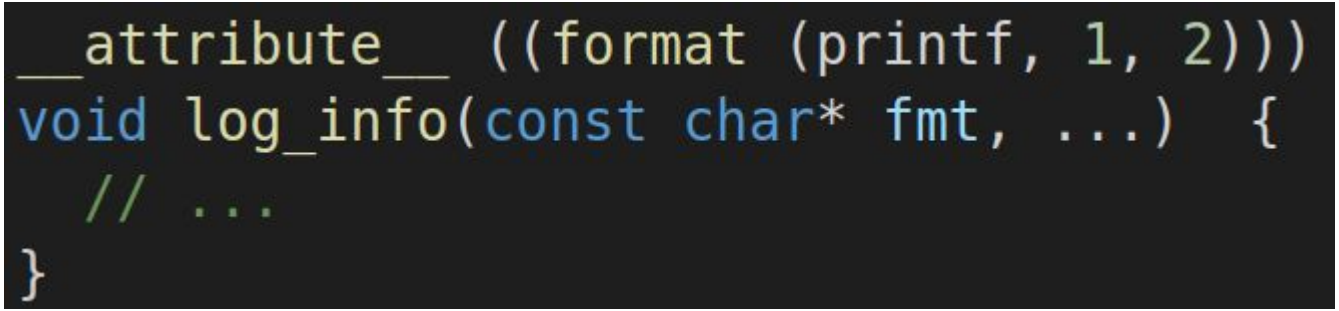
- Compilers will **NOT** emit **warnings** for these



```
void main() {
  char buf[BUF_SZ] = {0};
  fgets(buf, BUF_SZ - 1, stdin);
  log_info("buf: %s\n", buf);

  // VULN: No warning
  log_info(buf);
}
```

Format string vulnerabilities

- Unless we add the **function attribute “format”**
- *__attribute__ (format ([printf|scanf|strftime|strfmon], string-index, first-to-check))*



```
__attribute__ ((format (printf, 1, 2)))
void log_info(const char* fmt, ...)  {
  // ...
}
```

Format string vulnerabilities - exploitability

- The “**%n**” (or “**%hhn**”) format **writes** the number of chars written so far
- “**%6$n**” will write this to the 6th argument (**positional arguments**)

- This can be used to achieve a **write-what-where** and the **RCE**

Format string vulnerabilities - exploitability

- Example payload:

  %2044c%10$hn%38912c%11$hn

Format string vulnerabilities - mitigations

- Windows disables “**%n**” by default:
  - To enable you would have to call `int _set_printf_count_output(int enable);` explicitly

Format string vulnerabilities - mitigations

- On linux there is **FORTIFY_SOURCE**:
  - need to use “-O1” or more when compiling to enable it

- **FORTIFY_SOURCE:**
  - **Runtime** check
  - Format strings containing “**%n**” may **NOT** be located in a **writeable address**
  - When using positional parameters, all arguments within the range must be consumed. So to use %7$x, you must also use 1,2,3,4,5 and 6.

Format string vulnerabilities - exploitability

- We can still use to **leak memory** addresses (and bypass ASLR)

Modeling format string vulns

Modeling format string vulns

- The **format argument** has to be a **constant** address inside a **read-only** section

Modeling format string vulns

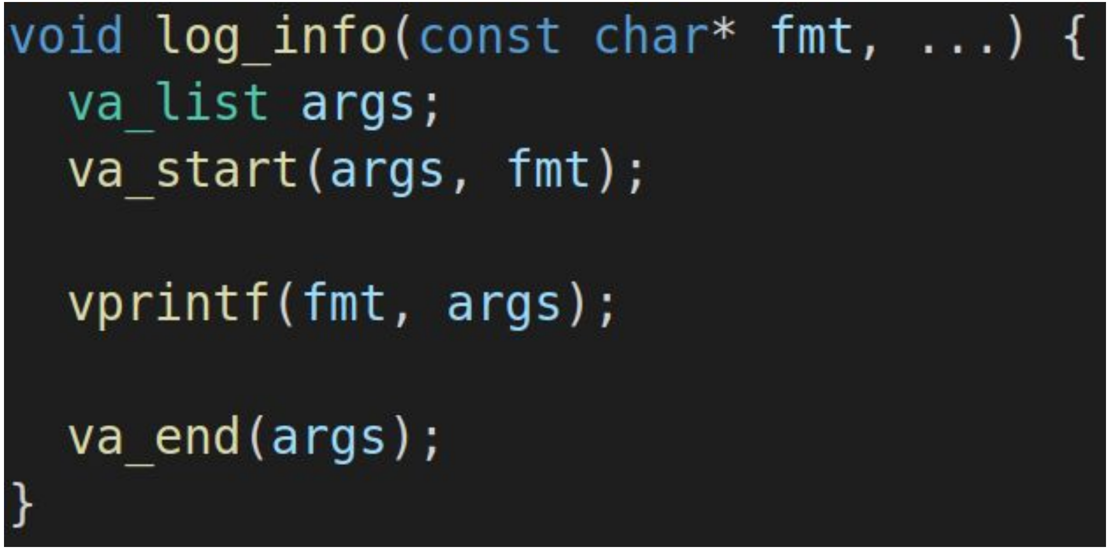
- **printf(“Hello”)** -> string comes from the **.rodata** section
- **printf(buf)** -> if buf is a **stack** or **heap** variable -> **not constant**
- **printf(buf)** -> if buf is a **global** variable -> **constant,** but **not read-only**

Modeling format string vulns

- Very **simple** and basically **what compilers do**
- How can we find the ones the compiler won’t warn us about?

Modeling format string vulns

- We need to find **printf-like functions** automatically (compilers wouldn't emit warnings)
- If the fmt parameter comes from a function argument -> **printf-like function**

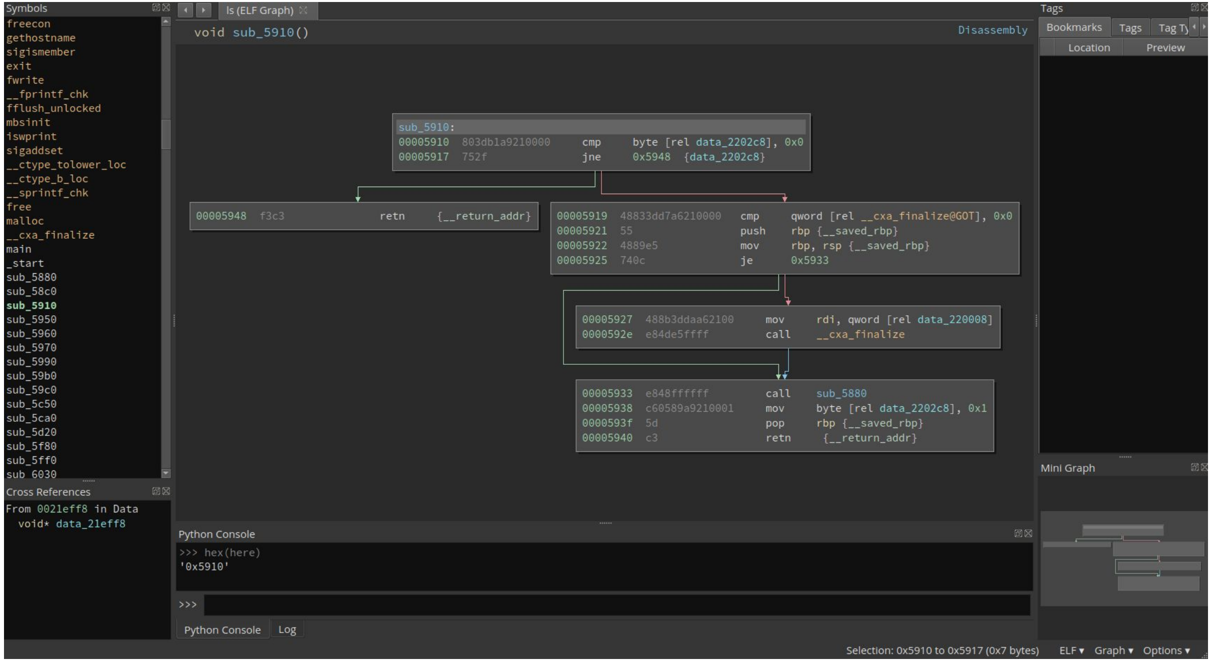


```
void log_info(const char* fmt, ...) {
  va_list args;
  va_start(args, fmt);

  vprintf(fmt, args);

  va_end(args);
}
```

Binary Ninja

Binary Ninja

- RE tool
  - PE, MachO, ELF, raw
  - x86, x64, arm, MIPS, PPC, ...
- Program analysis tool
  - great api
  - easy to script
  - headless (with the comercial license)



```
freecon
gethostname
sigismember
exit
fwrite
__fprintf_chk
fflush_unlocked
mbsinit
iswprint
sigaddset
__ctype_tolower_loc
__ctype_b_loc
__sprintf_chk
free
malloc
__cxa_finalize
main
_start
sub_5880
sub_58c0
sub_5910
sub_5950
sub_5960
sub_5970
sub_5990
sub_59b0
sub_59c0
sub_5c50
sub_5ca0
sub_5d20
sub_5f80
sub_5ff0
sub_6030
```

```
From 0021eff8 in Data
  void* data_21eff8
```

```
void sub_5910()

sub_5910:
00005910  803db1a9210000     cmp     byte [rel data_2202c8], 0x0
00005917  752f               jne     0x5948  {data_2202c8}

00005948  f3c3               retn     {__return_addr}

00005919  48833dd7a6210000   cmp     qword [rel __cxa_finalize@GOT], 0x0
00005921  55                 push    rbp {__saved_rbp}
00005922  4889e5             mov     rbp, rsp {__saved_rbp}
00005925  740c               je      0x5933

00005927  488b3ddaa62100     mov     rdi, qword [rel data_220008]
0000592e  e84de5ffff         call    __cxa_finalize

00005933  e848ffffff         call    sub_5880
00005938  c60589a9210001     mov     byte [rel data_2202c8], 0x1
0000593f  5d                 pop     rbp {__saved_rbp}
00005940  c3                 retn     {__return_addr}
```

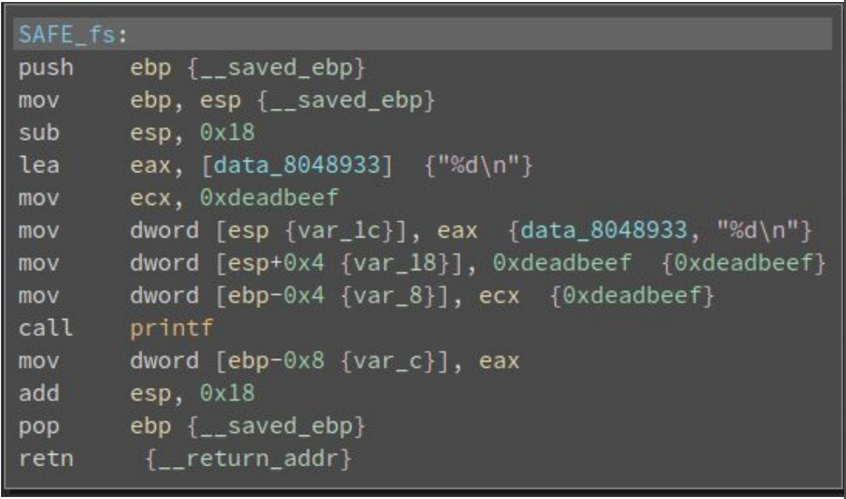
```
>>> hex(here)
'0x5910'
>>>
```

Selection: 0x5910 to 0x5917 (0x7 bytes)

Binary Ninja - Intermediate languages

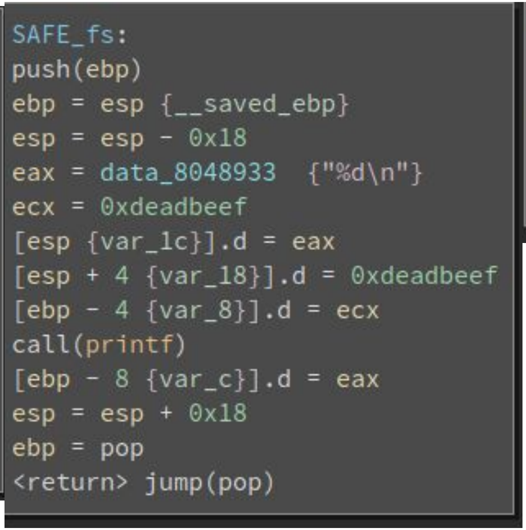
- Has several intermediate languages: **LLIL**, **MLIL**, (**HLIL** coming soon)
- ILs = analysis is **arch agnostic**

Disassembly



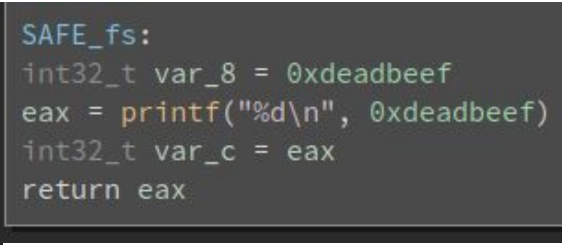
```
SAFE_fs:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
sub     esp, 0x18
lea     eax, [data_8048933]  {"%d\n"}
mov     ecx, 0xdeadbeef
mov     dword [esp {var_1c}], eax  {data_8048933, "%d\n"}
mov     dword [esp+0x4 {var_18}], 0xdeadbeef  {0xdeadbeef}
mov     dword [ebp-0x4 {var_8}], ecx  {0xdeadbeef}
call    printf
mov     dword [ebp-0x8 {var_c}], eax
add     esp, 0x18
pop     ebp {__saved_ebp}
retn     {__return_addr}
```

LLIL



```
SAFE_fs:
push(ebp)
ebp = esp {__saved_ebp}
esp = esp - 0x18
eax = data_8048933  {"%d\n"}
ecx = 0xdeadbeef
[esp {var_1c}].d = eax
[esp + 4 {var_18}].d = 0xdeadbeef
[ebp - 4 {var_8}].d = ecx
call(printf)
[ebp - 8 {var_c}].d = eax
esp = esp + 0x18
ebp = pop
<return> jump(pop)
```

MLIL



```
SAFE_fs:
int32_t var_8 = 0xdeadbeef
eax = printf("%d\n", 0xdeadbeef)
int32_t var_c = eax
return eax
```

Single static assignment (SSA) form

Normal form

```
a = 10
b = 20
a = a + b
```

SSA form

```
a_1 = 10
b_1 = 20
a_2 = a_1 + b_1
```

Single static assignment (SSA) form

Normal form

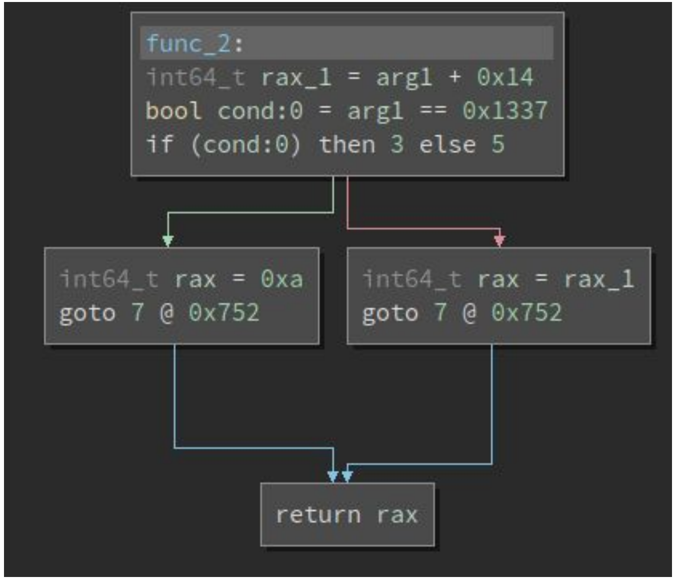
```
def func(a):
    if a == 1337:
        b = 10
    else:
        b = a + 20

    return b
```

SSA form

```
def func(a_0):
    if a_0 == 1337:
        b_1 = 10
    else:
        b_2 = a_0 + 20
    b_3 = Φ(b_1, b_2)
    return b_3
```

Single static assignment (SSA) form

- SSA makes it easy to trace the **uses and definitions of a variable** in a program

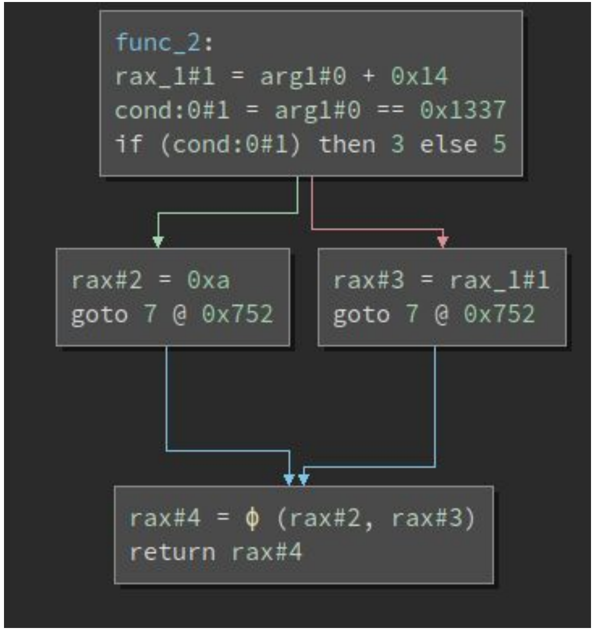


```
func_2:
int64_t rax_1 = arg1 + 0x14
bool cond:0 = arg1 == 0x1337
if (cond:0) then 3 else 5
```

```
int64_t rax = 0xa
goto 7 @ 0x752
```

```
int64_t rax = rax_1
goto 7 @ 0x752
```

```
return rax
```



```
func_2:
rax_1#1 = arg1#0 + 0x14
cond:0#1 = arg1#0 == 0x1337
if (cond:0#1) then 3 else 5
```

```
rax#2 = 0xa
goto 7 @ 0x752
```

```
rax#3 = rax_1#1
goto 7 @ 0x752
```

```
rax#4 = ϕ (rax#2, rax#3)
return rax#4
```

Using binja to model format string vulns

Using binja to model format string vulns - overview

1. **Load** all known **printf-like functions**
2. Iterate over the **xrefs,** analysing the origins of the format argument:
   a. origin is an **argument** -> add to the list of **printf-like functions**
   b. origin is an **contant** and **read-only** address -> **SAFE!**
   c. origin is a **known safe function** -> **SAFE!**
   d. Otherwise -> **VULN!**

Using binja to model format string vulns

- **Step 1**: Load all known printf-like functions

```
# int printf(const char *format, ...);
printf 0

# int fprintf(FILE *stream, const char *format, ...);
fprintf 1

# int dprintf(int fd, const char *format, ...);
dprintf 1

# int sprintf(char *str, const char *format, ...);
sprintf 1

# int snprintf(char *str, size_t size, const char *format, ...);
snprintf 2
(...)
```

Using binja to model format string vulns

- **Step 2.1**: Iterate over each xref

```
to_visit = PrintfLikeFunction.load_all()

while to_visit:
    printf_like_func = to_visit.pop(0)

    sym = self.bv.get_symbol_by_raw_name(printf_like_func.name)
    if not sym: # this function is not present in the binary
        continue

    for ref in self.bv.get_code_refs(sym.address):
        (...)
```

Using binja to model format string vulns

- **Step 2.2**: Get the format argument of the xref

```
mlil_instr = ref.function.get_low_level_il_at(ref.address).medium_level_il
if mlil_instr.operation not in (MLILOperation.MLIL_CALL, MLILOperation.MLIL_TAILCALL):
    # We don’t want to analyze cases where the address of the function is being written and not called (MLIL_SET_VAR)
    continue

fmt_param = mlil_instr.ssa_form.params[printf_like_func.parameter_index]
```

Using binja to model format string vulns

- **Step 2.3**: Get the **origins** for the **format argument**

```
if fmt_param.operation in (MLILOperation.MLIL_CONST, MLILOperation.MLIL_CONST_PTR):
    # mark as const
    var_origins = [VarOriginConst(fmt_param.constant)]
elif fmt_param.operation in (MLILOperation.MLIL_VAR_SSA, MLILOperation.MLIL_VAR_ALIASED):
    # create a backwards slice, starting from the fmt arg and tracing all the way back to its origin(s)
    var_origins = get_var_origins(fmt_param) # detailed code omitted for simplicity
```

- Origins can be:
    - VarOriginParameter
    - VarOriginConst
    - VarOriginCallResult
    - VarOriginUnknown

Using binja to model format string vulns

- **Step 2.4**: With the origins, determine if the call is **safe**:

```python
if isinstance(orig, VarOriginParameter):
    to_visit.append(PrintfLikeFunction(ref.function.name, orig.parameter_idx))
elif isinstance(orig, VarOriginConst) and self.is_addr_read_only(orig.const):
    safe_origins.append(orig)
elif isinstance(orig, VarOriginCallResult) and orig.func_name in self.safe_functions:
    # We accept that 'dcgettext' is safe because you need root to control the translation files
    safe_origins.append(orig)
else:
    vuln_origins.append(orig)
```

“gettext” family of functions

- Used for **translation**
- If we could control *“/usr/share/locale/<lang>/LC_MESSAGES”* -> trigger format strings
- But, files are owned by **root** -> we consider these safe

DEMO

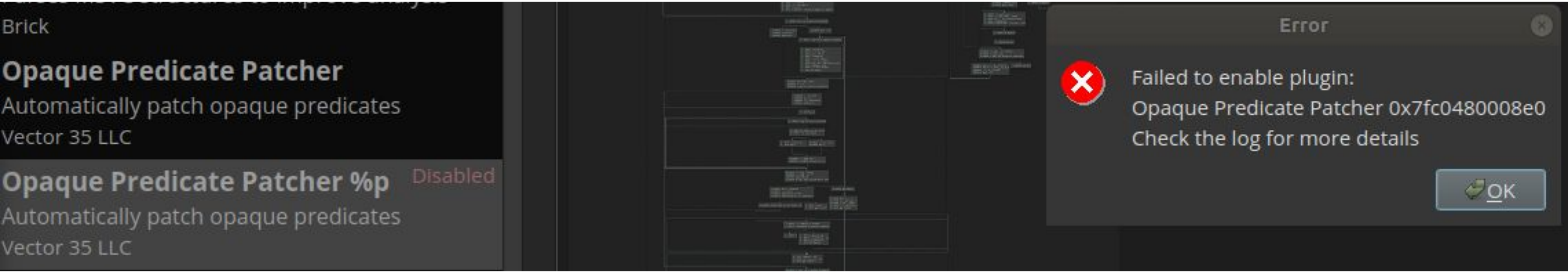
What this plugin won’t find

- When **binja analysis fails** and xrefs are missed
- When the call is an **indirect call** -> via a vtable or function pointer

Fun fact

Fun fact

- Was trying to find edge cases against **complex software**
- Tested with **binary ninja**
- Found a vulnerability when displaying the **plugin name**



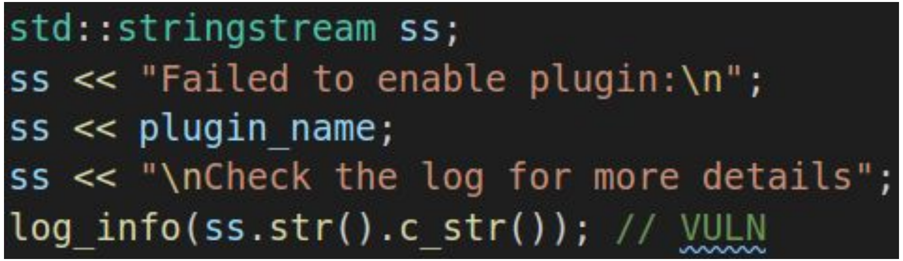
Brick

Opaque Predicate Patcher
Automatically patch opaque predicates
Vector 35 LLC

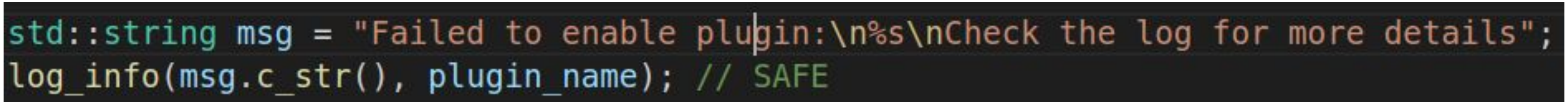
Opaque Predicate Patcher %p Disabled
Automatically patch opaque predicates
Vector 35 LLC

Failed to enable plugin:
Opaque Predicate Patcher 0x7fc0480008e0
Check the log for more details

Fun fact - vuln code and fix



```
std::stringstream ss;
ss << "Failed to enable plugin:\n";
ss << plugin_name;
ss << "\nCheck the log for more details";
log_info(ss.str().c_str()); // VULN
```



```
std::string msg = "Failed to enable plugin:\n%s\nCheck the log for more details";
log_info(msg.c_str(), plugin_name); // SAFE
```

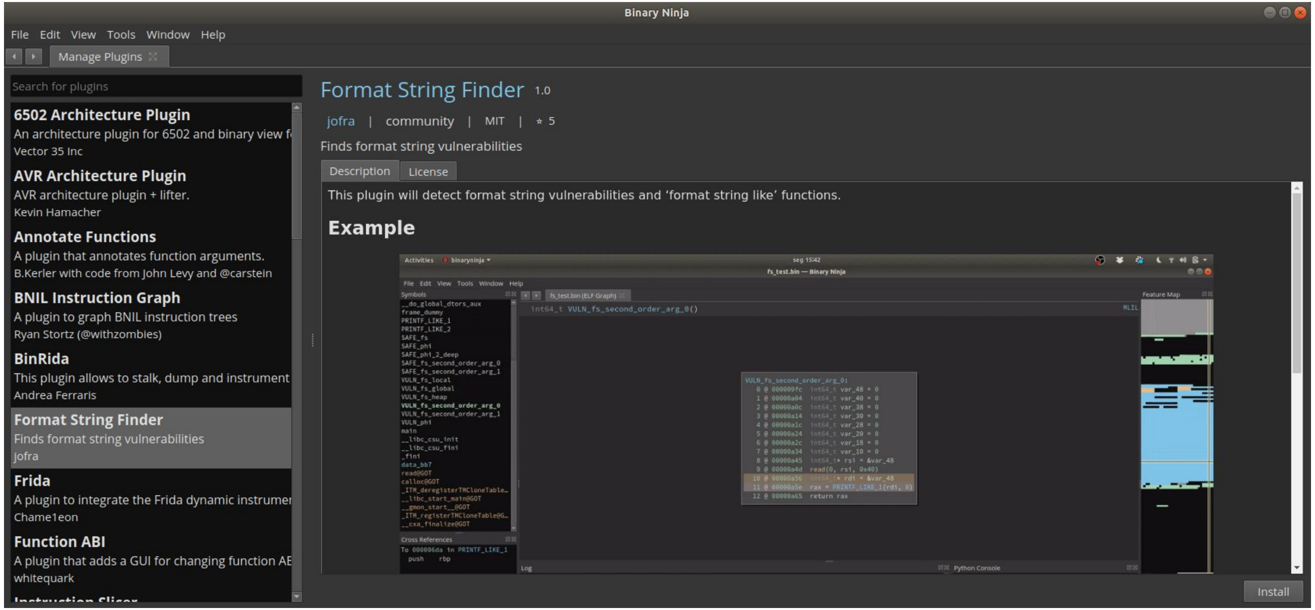
Fun fact - security impact

- **No security impact**
- Has FORTIFY_SOURCE -> not exploitable
- Plugins are already code anyway

Final thoughts

Final thoughts

- https://github.com/Vasco-jofra/format-string-finder-binja
- Can also find it in the plugin manager:



6502 Architecture Plugin
An architecture plugin for 6502 and binary view f
Vector 35 Inc

AVR Architecture Plugin
AVR architecture plugin + lifter.
Kevin Hamacher

Annotate Functions
A plugin that annotates function arguments.
B.Kerler with code from John Levy and @carstein

BNIL Instruction Graph
A plugin to graph BNIL instruction trees
Ryan Stortz (@withzombies)

BinRida
This plugin allows to stalk, dump and instrument
Andrea Ferraris

Format String Finder
Finds format string vulnerabilities
jofra

Frida
A plugin to integrate the Frida dynamic instrumer
Chame1eon

Function ABI
A plugin that adds a GUI for changing function AE
whitequark

Format String Finder 1.0

jofra | community | MIT | ⭑ 5

Finds format string vulnerabilities

This plugin will detect format string vulnerabilities and 'format string like' functions.

Example

```
__do_global_dtors_aux
frame_dummy
PRINTF_LIKE_1
PRINTF_LIKE_2
SAFE_fs
SAFE_phi
SAFE_phi_2_deep
SAFE_fs_second_order_arg_0
SAFE_fs_second_order_arg_1
VULN_fs_local
VULN_fs_global
VULN_fs_heap
VULN_fs_second_order_arg_0
VULN_fs_second_order_arg_1
VULN_phi
main
__libc_csu_init
__libc_csu_fini
_fini
data_bb7
read@GOT
calloc@GOT
_ITM_deregisterTMCloneTable…
__libc_start_main@GOT
__gmon_start__@GOT
_ITM_registerTMCloneTable@G…
__cxa_finalize@GOT
```

```
To 000006da in PRINTF_LIKE_1
  push    rbp
```

```
int64_t VULN_fs_second_order_arg_0()
```

```
VULN_fs_second_order_arg_0:
   0 @ 000009fc  int64_t var_48 = 0
   1 @ 00000a04  int64_t var_40 = 0
   2 @ 00000a0c  int64_t var_38 = 0
   3 @ 00000a14  int64_t var_30 = 0
   4 @ 00000a1c  int64_t var_28 = 0
   5 @ 00000a24  int64_t var_20 = 0
   6 @ 00000a2c  int64_t var_18 = 0
   7 @ 00000a34  int64_t var_10 = 0
   8 @ 00000a45  int64_t* rsi = &var_48
   9 @ 00000a4d  read(0, rsi, 0x40)
  10 @ 00000a56  int64_t* rdi = &var_48
  11 @ 00000a5e  rax = PRINTF_LIKE_1(rdi, 0)
  12 @ 00000a65  return rax
```

Final thoughts

- Hope you learned something about how to model vulnerabilities in binary ninja
- Join the binja slack -> awesome community

Thanks!

QUESTIONS?