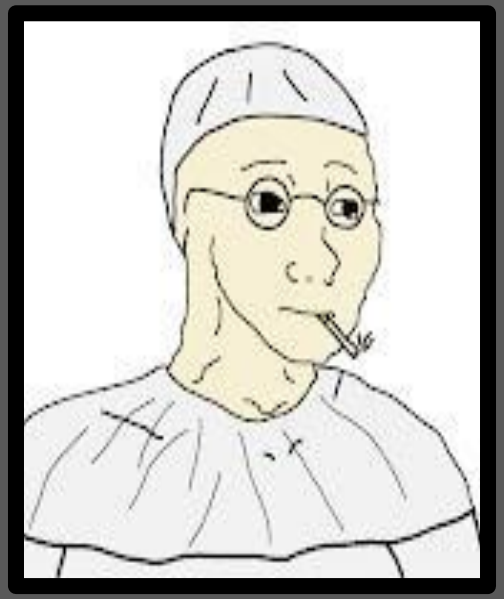
Automated program analysis

1) Whoami
2) What is program analysis?
3) Why does it matter?
4) Static and dynamic analysis
5) Problems and limitations
6) Creating your own tooling
7) Automated vulnerability discovery

WHOAMI

Captain @
Equivalent XCHG

VR/RE Tooling
Automation



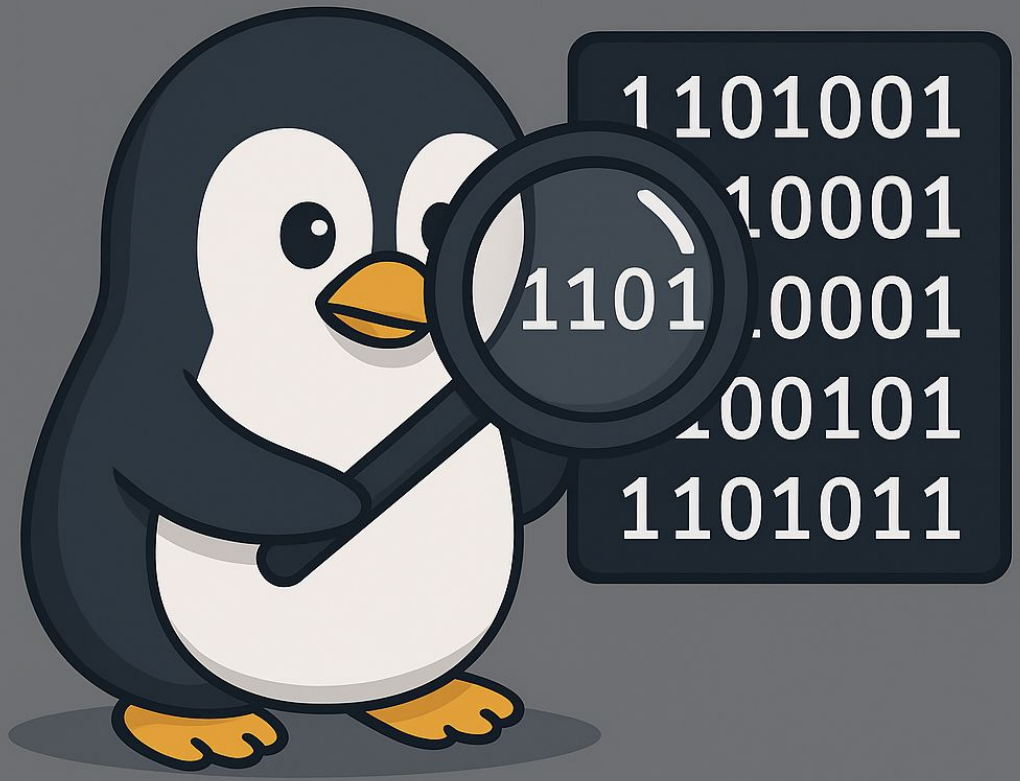
Vulnerability
researcher and
reverse engineer

Pwnpope on:

Automated program analysis trophy case

- AFL trophy case
  - Thousands of bugs and vulnerabilities have been found with the AFL suite of fuzzers since its creation in 2013.
- Zoncolan (Meta/Facebook)
  - Found 1,500+ security issues in 2018 alone across Facebook's 100M+ line codebase using custom static analysis.
- Mayhem (DARPA Cyber Grand Challenge winner)
  - Performed autonomous vulnerability detection and patching in real-time, winning a $2M prize.
- Google OSS-Fuzz
  - Continuously fuzzes 1,000+ open-source projects.
  - Reported over 50,000 bugs and 10,000 CVEs since 2016.
- Semmle/QI (now GitHub CodeQL)
  - Used to detect thousands of bugs via static analysis in open-source projects.
  - Played a role in detecting CVE-2020-0601 (Windows CryptoAPI vuln).

What is program analysis?



Understanding how code behaves

- Analyzing binaries or source code for bugs, optimizations, vulnerabilities, etc
- Can be static (e.g., slicing, CFGs, IR inspection) or dynamic (e.g., sys call tracing)
- Used in compilers, reverse engineering, security auditing, etc.
- Built on structured code representations like control flow graphs and data flow analysis

Binary Ninja

- Great decompiler used for reverse engineering compiled binaries.
- Great community support and documentation
- Amazing UI
- Feature rich, powerful and well documented API
- Amazing plugin store and super easy to make plugins
- Fairly priced

Why does it matter?

- Faster Bug Hunting
- Automated reverse engineering
- Specialized analysis
- High speed auditing
- Continuous and repeatable analysis
- Empowers custom tooling and workflows

And much much more…

Static and dynamic analysis

Static

Analysis WITHOUT having to run the program, for example, variable use-def chains.

Semgrep

Dynamic

Analysis while the program is running, for example, debugging, fuzzing, sys-call tracing, function hooking, etc.

Problems and limitations

- Path explosion
- Loops and recursion
- Environment modeling (e.g., handling library or system calls)
- Implementation limitations (e.g., solver constraints or timeouts)
- Heap modeling
- Indirect control flow
- Obfuscated or packed code
- Encrypted or runtime decrypted code

Wait but I thought that automated program analysis was the end all be all?

Creating your own tooling

Ez pz, right?

Wrong...

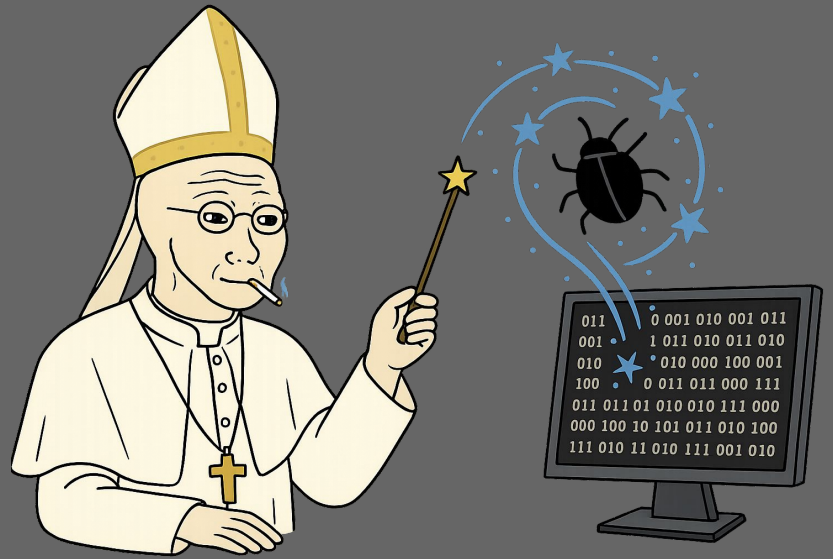
Problems when creating your own tools

- **Creating program analysis tooling is extremely difficult**
  - Takes many intermediate and advanced topics from graph theory to compiler design and more to wrap everything together and make a “good” tool
  - Also depending on your tooling you need to understand IRs, control/data flow, lifetimes, taint models, and edge cases across architectures, etc.
- **Tooling is often extremely specialized with a low threshold for leeway (e.g., symbolic execution engines**
  - Symbolic engines work well in constrained domains, but blow up with recursion, dynamic memory, or real-world libraries.
- **Cross-platform challenges and cross-architecture challenges**
  - Tooling must account for calling conventions, ABI quirks, packed binaries, and kernel-level behavior.
- **Debugging your analysis is hell**
  - Analysis bugs aren’t obvious – they're silent failures.
  - You’ll trace IR, logs, and slices for hours just to find one missing condition.

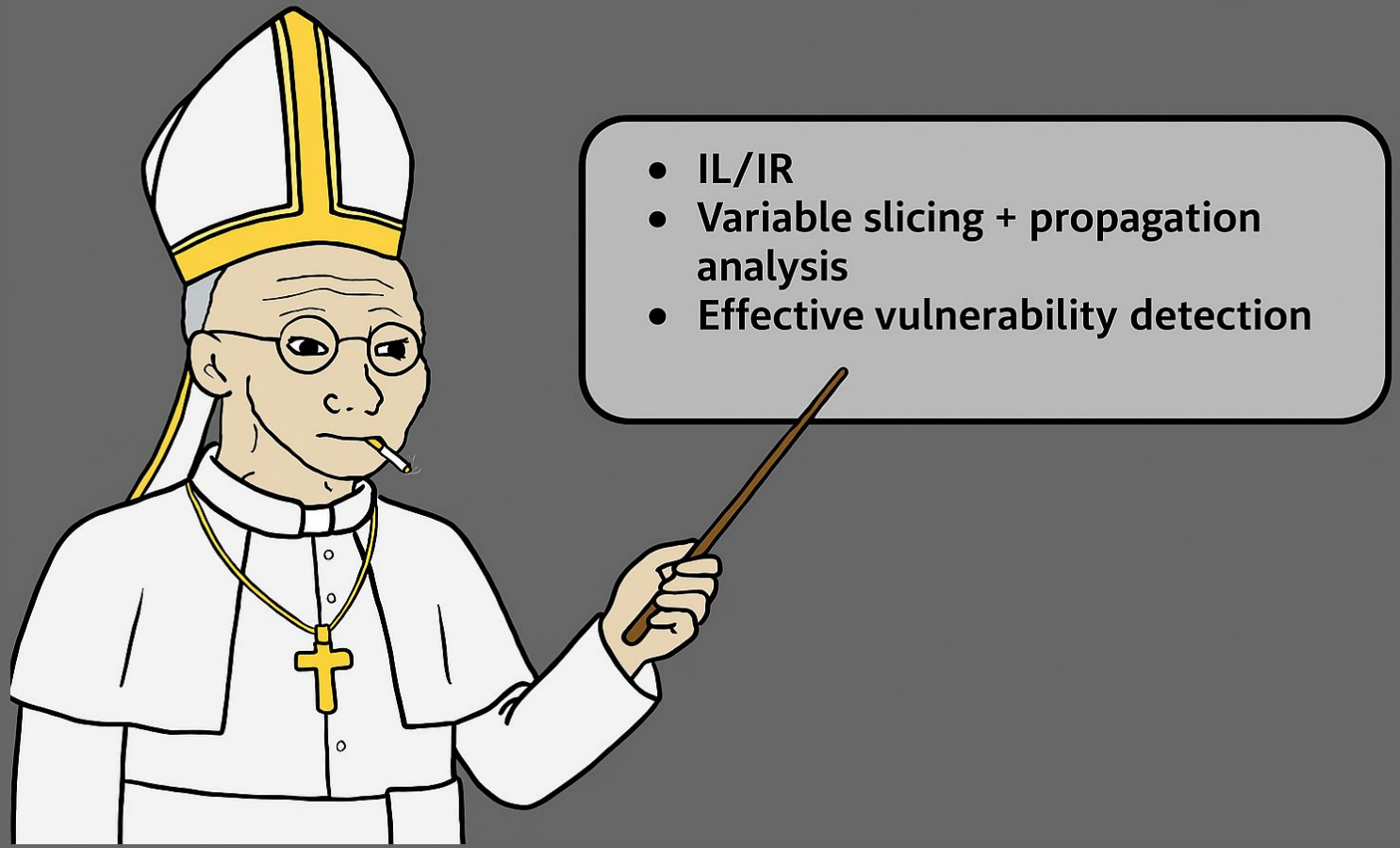
But who cares about the difficulty? I just wanna build my own tool for doing cool analysis and potentially find some bugs while i’m at it.

Automated vulnerability discovery

Binja API + Taint analysis + Magic = Bugs



First we need to understand a few things...



IR/IL

Intermediate language or intermediate representation is a lifted representation of a given programs bytecodes often determined via the disassembly, the terms IL/IR are used interchangeably and are essentially synonymous.

IR/IL 1/3

Source code

Disassembly

```
215  void do_free(char *buffer) {
216      printf("[do_free] Freeing buffer...\n");
217      free(buffer);
218  }
```

```
004015ad    int64_t free_buffer(char* buffer)

004015ad  f30f1efa          endbr64
004015b1  55                push    rbp {__saved_rbp}
004015b2  4889e5            mov     rbp, rsp {__saved_rbp}
004015b5  4883ec10          sub     rsp, 0x10
004015b9  48897df8          mov     qword [rbp-0x8 {var_10}], rdi
004015bd  488d05740c0000    lea     rax, [rel data_402238]
004015c4  4889c7            mov     rdi, rax  {data_402238, "Buffer passed to free_buffer, wi…"}
004015c7  e814fbffff        call    puts
004015cc  488b45f8          mov     rax, qword [rbp-0x8 {var_10}]
004015d0  4889c7            mov     rdi, rax
004015d3  e8f8faffff        call    free
004015d8  90                nop
004015d9  c9                leave   {__saved_rbp}
004015da  c3                retn    {__return_addr}
```

IR/IL 2/3

LLIL

```
int64_t free_buffer(char* buffer)

push(rbp)
rbp = rsp {__saved_rbp}
rsp = rsp - 0x10
[rbp - 8 {var_10}].q = rdi
rax = &data_402238  {"Buffer passed
rdi = rax
call(puts)
rax = [rbp - 8 {var_10}].q
rdi = rax
call(free)
rsp = rbp
rbp = pop
<return> jump(pop)
```

MLIL

```
int64_t free_buffer(char* buffer)

var_10 = buffer
puts(str: "Buffer passed to free_buf
rax = var_10
rdi = rax
rax_1 = free(mem: rdi)
return rax_1
```

HLIL

```
int64_t free_buffer(char* buffer)

puts(str: "Buffer passed to free_buffer, wi…")
return free(mem: buffer)
```

IR/IL 3/3

Pseudo C

```
int64_t free_buffer(char* buffer)

{
    puts("Buffer passed to free_buffer, wi…");
    return free(buffer);
}
```

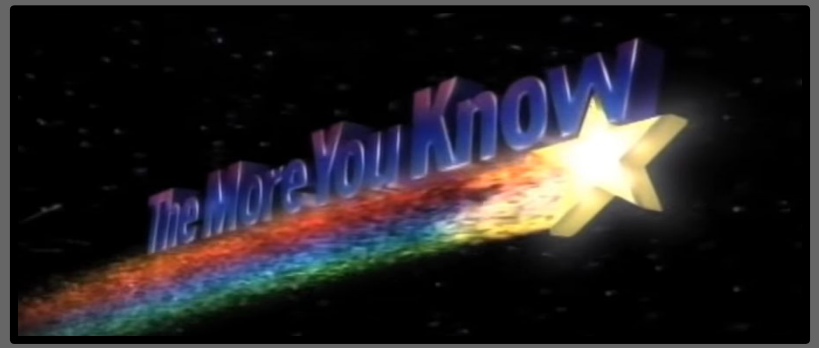
Variable slicing and propagation analysis 1/2

Variable slicing

Finding each point a variable is used.

Propagation analysis

The process of finding all spots a variable propagates too, meaning other variables, memory, etc.



Variable slicing and propagation analysis 2/2

Variable slicing

```
int64_t level_four()

buf = malloc(bytes: 0x64)
buffer_a = buf
if (buffer_a != 0) then 3 @ 0x401513 else 29 @ 0x4014ff

rax_1 = 0
printf(format: "Enter a string for level_four (b…")
rdx_1 = [&__TMC_END__]
rax_2 = buffer_a
rdi_1 = rax_2
fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
rax_3 = buffer_a
rsi_1 = rax_3
rax_4 = 0
printf(format: "Buffer A before freeing: %s\n", rsi_1)
rax_5 = buffer_a
rdi_2 = rax_5
free(mem: rdi_2)
rax_6 = buffer_a
buffer_b = rax_6
rax_7 = 0
printf(format: "Buffer B after passing freed buf…")
rdx_2 = [&__TMC_END__]
rax_8 = buffer_b
rdi_3 = rax_8
fgets(buf: rdi_3, n: 0x64, fp: rdx_2)
rax_9 = buffer_b
rsi_2 = rax_9
rax_10 = 0
rax = printf(format: "Content from buffer_b after UAF:…", rsi_2)
goto 31 @ 0x4015ac

rax = perror(s: "malloc failed")
goto 31 @ 0x4015ac

return rax
```

Propagation analysis

```
[[0x4014ea] buffer_a = buf -> buffer_a -> buf [Tainted],
 [0x401524] rax_2 = buffer_a -> buffer_a -> buf [Tainted],
 [0x40152d] rdi_1 = rax_2 -> rax_2 -> buffer_a [Tainted],
 [0x401530] fgets(rdi_1, 0x64, rdx_1) -> rdi_1 -> rax_2 [Tainted],
 [0x401535] rax_3 = buffer_a -> buffer_a -> buf [Tainted],
 [0x401539] rsi_1 = rax_3 -> rax_3 -> buffer_a [Tainted],
 [0x40154b] printf("Buffer A before freeing: %s\n", rsi_1) -> rsi_1 -> rax_3 [Tainted],
 [0x401550] rax_5 = buffer_a -> buffer_a -> buf [Tainted],
 [0x401554] rdi_2 = rax_5 -> rax_5 -> buffer_a [Tainted],
 [0x401557] free(rdi_2) -> rdi_2 -> rax_5 [Tainted],
 [0x40155c] rax_6 = buffer_a -> buffer_a -> buf [Tainted],
 [0x401560] buffer_b = rax_6 -> rax_6 -> buffer_a [Tainted],
 [0x40157f] rax_8 = buffer_b -> buffer_b -> rax_6 [Tainted],
 [0x401588] rdi_3 = rax_8 -> rax_8 -> buffer_b [Tainted],
 [0x40158b] fgets(rdi_3, 0x64, rdx_2) -> rdi_3 -> rax_8 [Tainted],
 [0x401590] rax_9 = buffer_b -> buffer_b -> rax_6 [Tainted],
 [0x401594] rsi_2 = rax_9 -> rax_9 -> buffer_b [Tainted],
 [0x4015a6] rax = printf("Content from buffer_b after UAF:…", rsi_2) -> rsi_2 -> rax_9 [Tainted]]
```

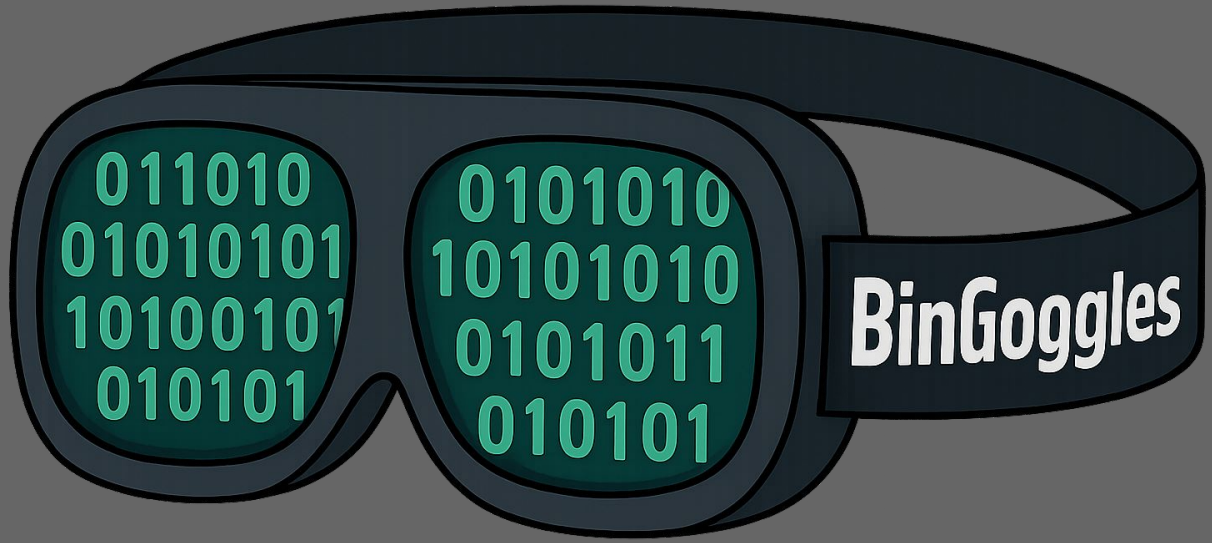
Effective vulnerability detection

Questions to answer during development

- How can I reduce false positives?
- What are the rule(s) that I need to create in my analysis that determines that the vulnerability exists?
- What assumptions does my analysis rely on?
- How am I gathering the data needed for my analysis?

And trust me there will be plenty more questions that spawn up whilst creating your tooling.

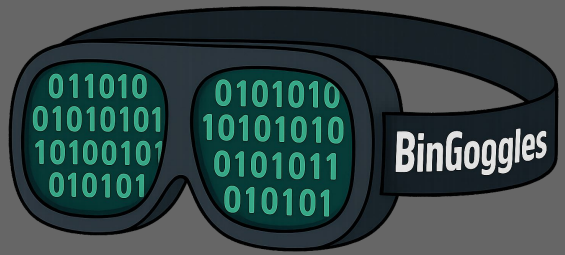
Introducing BinGoggles



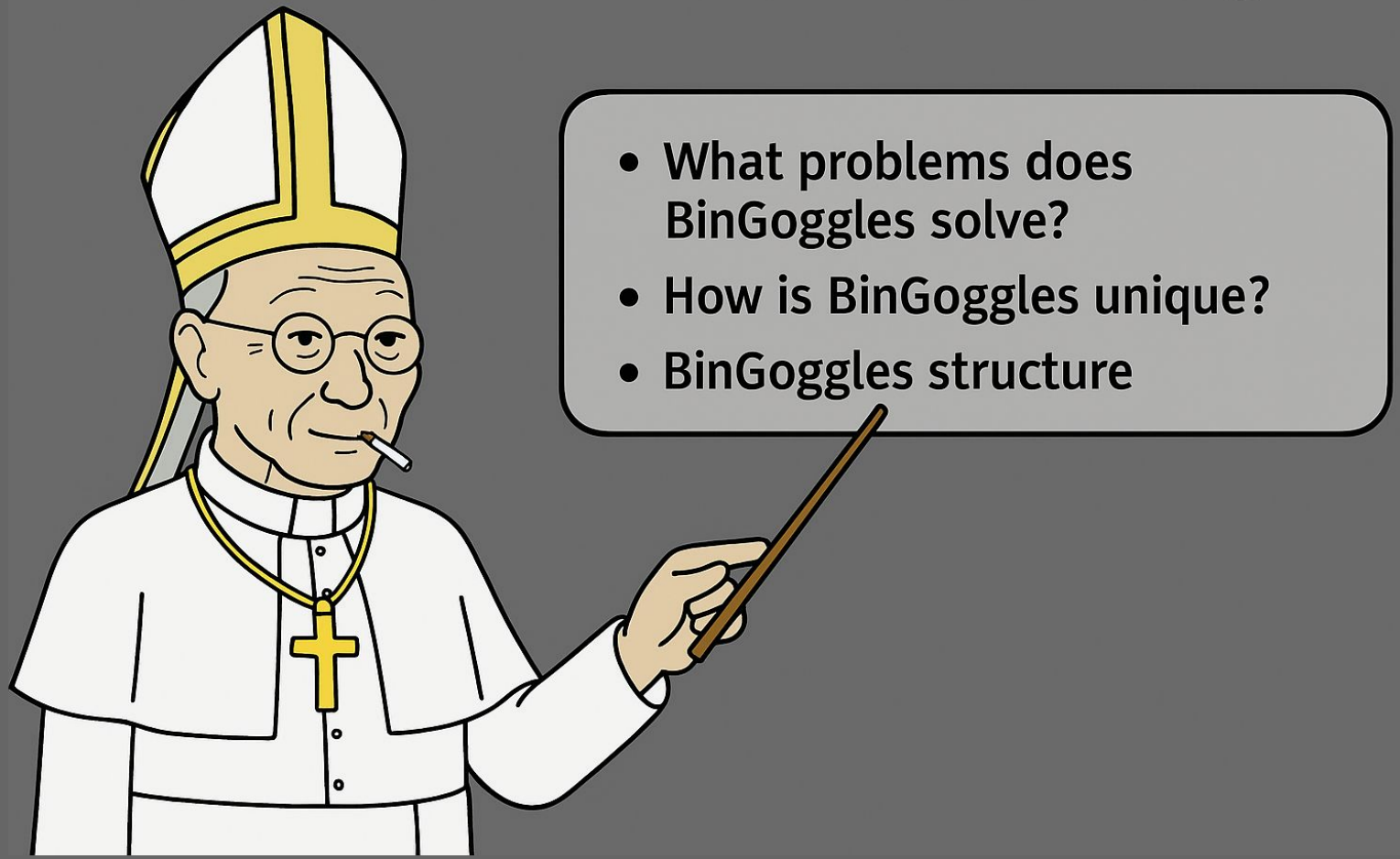
What is BinGoggles?

BinGoggles is an API utilizing binja to gather a variables flow in a program statically.

- **Library, architecture, and platform agnostic**
- **Extremely portable**
- **Intraprocedural and interprocedural analysis**
- **Works on all types of variables**

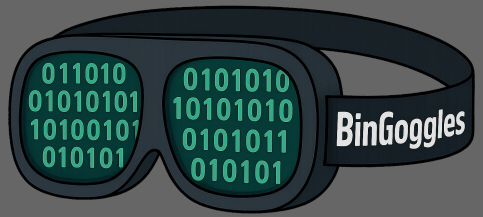


First we need to understand a few things...



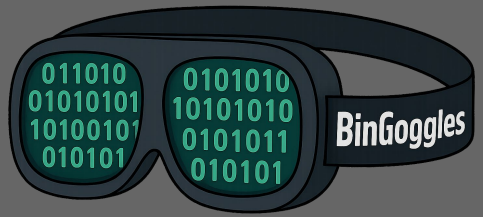
What problems does BinGoggles solve?

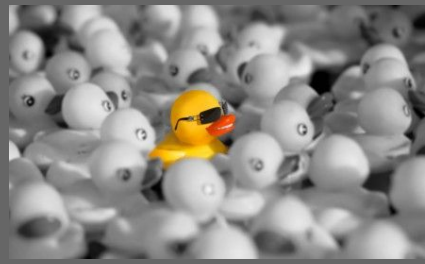
- Library, architecture, and platform agnostic
- Intraprocedural and interprocedural analysis
- Works on all types of variables from normal variables to struct members, globals, and function parameters
- Both backwards and forwards analysis
- Extremely easy to use
- Works with both binja headless and hugsy headless



How is BinGoggles unique?

- Great for everything from RTOS devices that may have little to no symbols to medium to large binaries with symbols or without symbols
- Complete static analysis of a variable flow path
- Flexibility, wanna go backwards or forwards? Go ahead
- In-depth, what if a function has sub-function calls that use the variable that we're tracing, BinGoggles has your back





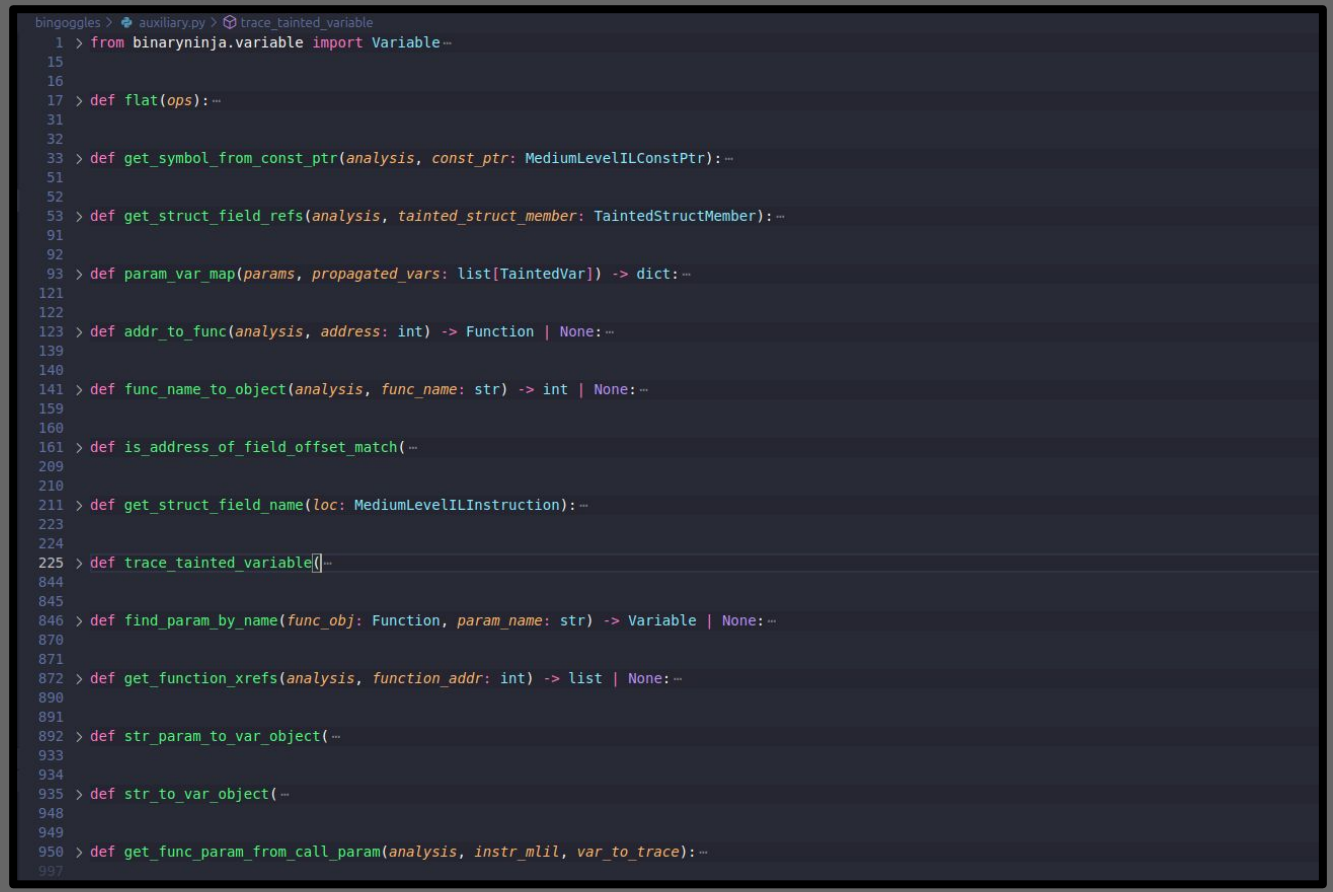
BinGoggles Structure 1/3

bingoggles/vfa.py

```
19  class Analysis:
20      def __init__(self, binaryview, verbose=None, libraries_mapped: dict = None):
21          self.bv = binaryview
22          self.verbose = verbose
23          self.libraries_mapped = libraries_mapped
24
25          if self.verbose:
26              self.is_function_param_tainted_printed = False
27
28  >   def get_sliced_calls(…
79
80      @cache
81  >   def tainted_forward_slice(…
322
323     @cache
324 >   def tainted_backwards_slice(…
438
439     @cache
440 >   def complete_slice(…
637
638 >   def is_function_param_tainted(…
932
933 >   def is_function_imported(self, instr_mlil: MediumLevelILInstruction) -> bool:…
952
953 >   def analyze_imported_function(self, func_symbol):…
958
```

BinGoggles Structure 2/3

bingoggles/auxiliary.py



BinGoggles Structure 3/3

bingoggles/bingoggles_types.py

```
bingoggles > bingoggles_types.py > ...
  1 > from dataclasses import dataclass ⋯
 18
 19
 20 > class IL: ⋯
 24
 25
 26 > class TaintConfidence: ⋯
 30
 31   @dataclass
 32 > class TaintedGlobal: ⋯
 77
 78
 79   @dataclass
 80 > class TaintedVar: ⋯
125
126
127 > class TaintedLOC: ⋯
183
184
185 > class SliceType: ⋯
188
189
190 > class SlicingID: ⋯
193
194
195 > class OutputMode: ⋯
198
199
200   @dataclass
201 > class TaintedAddressOfField: ⋯
301
302
303 > class BGInit: ⋯
398
399
400 > class BGInitRpyc(BGInit): ⋯
464
465
466   #: TODO
467   @dataclass
468 > class TaintedStructMember: ⋯
516
517
518   @dataclass
519 > class TaintTarget: ⋯
534
535
536   @dataclass
```

Using BinGoggles 1/4

Picking a target variable

```c
#include <stdio.h>
#include <string.h>

int main() {
    int key_index = 5;
    char buf[11] = {0};
    buf[0] = 'H';
    buf[1] = 'E';
    buf[2] = 'L';
    buf[3] = 'L';
    buf[4] = 'O';
    buf[5] = ' ';
    buf[6] = 'P';
    buf[7] = 'O';
    buf[8] = 'P';
    buf[9] = 'E';

    char encrypted_buf[10];

    for (int i = 0; i < strlen(buf) - 1; i++) {
        encrypted_buf[i] = 0x50 ^ buf[i];
    }

    printf("Original Buffer: %s\nEncrypted Buffer: %s\n", buf, encrypted_buf);

    return 0;
}
```

ELF ▾ Linear ▾ Medium Level IL ▾

```
int32_t main()

  0 @ 00401196  rax = [fsbase + 0x28].q
  1 @ 0040119f  var_20 = rax
  2 @ 004011a3  rax_1 = 0
  3 @ 004011a5  key_index = 5
  4 @ 004011ac  buf[0].q = 0
  5 @ 004011b4  buf[7].d = 0
  6 @ 004011bb  __builtin_strncpy(dest: &buf:0, src: "HELLO POPE", n: 0xa)
  7 @ 004011e3  var_40 = 0
  8 @ 004011ea  goto 9 @ 0x401208

  9 @ 00401208  rax_8 = var_40
 10 @ 0040120b  rbx_1 = sx.q(rax_8)
 11 @ 00401212  rdi = &buf
 12 @ 00401215  rax_9 = strlen(rdi)
 13 @ 0040121a  rax_10 = rax_9 - 1
 14 @ 00401221  if (rbx_1 u< rax_10) then 15 @ 0x4011ec else 25 @ 0x401223

 15 @ 004011ec  rax_2 = var_40
 16 @ 004011ef  rax_3 = sx.q(rax_2)
 17 @ 004011f1  rax_4 = [&buf + rax_3].b
 18 @ 004011f6  rax_5 = rax_4 ^ 0x50
 19 @ 004011f9  rdx = rax_5
 20 @ 004011fb  rax_6 = var_40
 21 @ 004011fe  rax_7 = sx.q(rax_6)
 22 @ 00401200  [&i:3 + rax_7].b = rdx
 23 @ 00401204  var_40 = var_40 + 1
 24 @ 00401204  goto 9 @ 0x401208

 25 @ 00401223  rdx_1 = &i:3
 26 @ 0040122b  rsi = &buf
 27 @ 00401238  rax_11 = 0
 28 @ 0040123d  printf(format: "Original Buffer: %s\nEncrypted B…", rsi, rdx_1)
 29 @ 00401242  rax_12 = 0
 30 @ 00401247  rdx_2 = var_20
 31 @ 0040124b  temp0 = rdx_2
 32 @ 0040124b  temp1 = [fsbase + 0x28].q
 33 @ 0040124b  rdx_3 = rdx_2 - [fsbase + 0x28].q
 34 @ 00401254  if (temp0 == temp1) then 35 @ 0x401260 else 36 @ 0x401256

 35 @ 00401260  return 0

 36 @ 00401256  __stack_chk_fail()
```

Using Bingoggles 2/4

Giving your input to BinGoggles

```
63  def test_fwd_slice_var(bg_init, test_bin="./test/binaries/bin/test_mlil_store"):
64      bg = bg_init(
65          target_bin=abspath(test_bin),
66          libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
67          host="127.0.0.1",
68          port=18812,
69      )
70      bv, libraries_mapped = bg.init()
71
72      analysis = Analysis(binaryview=bv, verbose=True, libraries_mapped=libraries_mapped)
73      sliced_data, _, _ = analysis.tainted_forward_slice(
74          #   11 @ 00401212  rdi = &buf
75          target=TaintTarget(0x00401212, "rdi"),
76          var_type=SlicingID.FunctionVar,
77      )
78
```

Using BinGoggles 3/4

BinGoggles output

```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_fwd_slice_var
================================================================================ test session starts ================================================================================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Loading binary...    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:

tainted_forward_slice(self, target: list[int, str], var_type: SlicingID, output: OutputMode = OutputMode.Returned)
==================================================================================================================

is_function_param_tainted(self, function_node: int | Function, tainted_params: Variable | str | list[Variable])
-> uint64_t strlen(char const* arg1):[<SSAVariable: arg1 version 0>]
=================================================================================================================

is_function_param_tainted(self, function_node: int | Function, tainted_params: Variable | str | list[Variable])
-> int32_t printf(char const* format, ...):[None]
=================================================================================================================
Address | LOC | Target Variable | Propagated Variable | Taint Confidence
------------------------------------------------------------------------
11 [0x401212] rdi = &buf -> rdi -> buf [Tainted]
12 [0x401215] rax_9 = strlen(rdi) -> rdi -> buf [Tainted]
17 [0x4011f1] rax_4 = [&buf + rax_3].b -> buf -> None [Tainted]
18 [0x4011f6] rax_5 = rax_4 ^ 0x50 -> rax_4 -> buf [Tainted]
19 [0x4011f9] rdx = rax_5 -> rax_5 -> rax_4 [Tainted]
22 [0x401200] [&i:3 + rax_7].b = rdx -> rdx -> rax_5 [Tainted]
25 [0x401223] rdx_1 = &i:3 -> i -> None [Tainted]
26 [0x40122b] rsi = &buf -> buf -> None [Tainted]
28 [0x40123d] printf("Original Buffer: %s\nEncrypted B…", rsi, rdx_1) -> rsi -> buf [Tainted]
```

Using BinGoggles 4/4

BinGoggles output

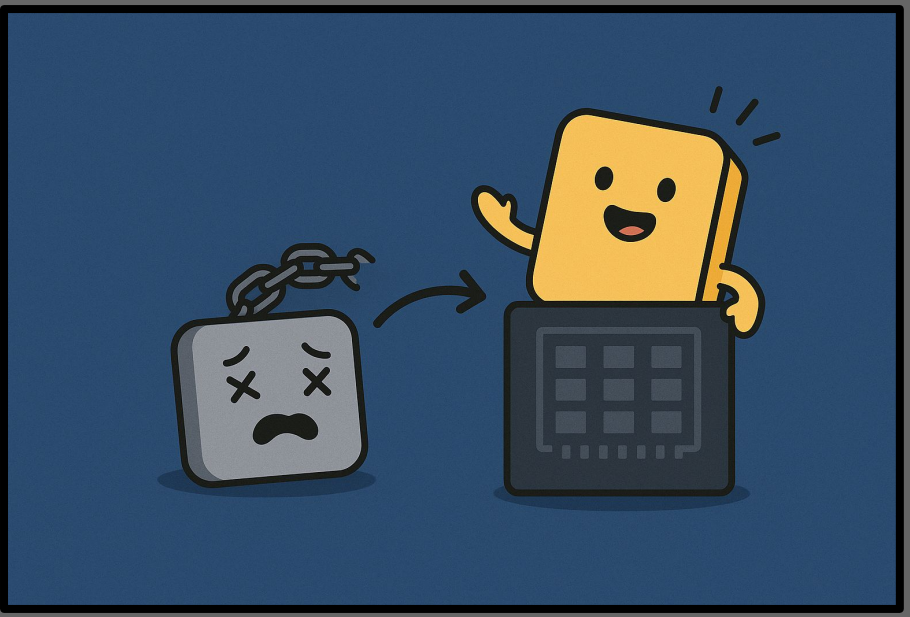
```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_fwd_slice_var
============================================================================================================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ──────────────────────────────────────── 100% 0:00:00
Loading binary...    ──────────────────────────────────────── 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
[[0x401212] rdi = &buf -> rdi -> buf [Tainted],
 [0x401215] rax_9 = strlen(rdi) -> rdi -> buf [Tainted],
 [0x4011f1] rax_4 = [&buf + rax_3].b -> buf -> None [Tainted],
 [0x4011f6] rax_5 = rax_4 ^ 0x50 -> rax_4 -> buf [Tainted],
 [0x4011f9] rdx = rax_5 -> rax_5 -> rax_4 [Tainted],
 [0x401200] [&i:3 + rax_7].b = rdx -> rdx -> rax_5 [Tainted],
 [0x401223] rdx_1 = &i:3 -> i -> None [Tainted],
 [0x40122b] rsi = &buf -> buf -> None [Tainted],
 [0x40123d] printf("Original Buffer: %s\nEncrypted B…", rsi, rdx_1) -> rsi -> buf [Tainted]]
[[rdi] -> [Tainted],
 [buf] -> [Tainted],
 [rax_4] -> [Tainted],
 [rsi] -> [Tainted],
 [rax_5] -> [Tainted],
 [rdx] -> [Tainted],
 [&i:3 + rax_7] -> [Byte(s) at reference + offset Tainted],
 [rdx_1] -> [Tainted]]
.
```

Analyzing the output

Creating something useful

So now we have to create a function that takes output from BinGoggles, analyzes it and tell us whether or not a use-after-free vulnerability exists in the variable path.

What's a use-after-free?

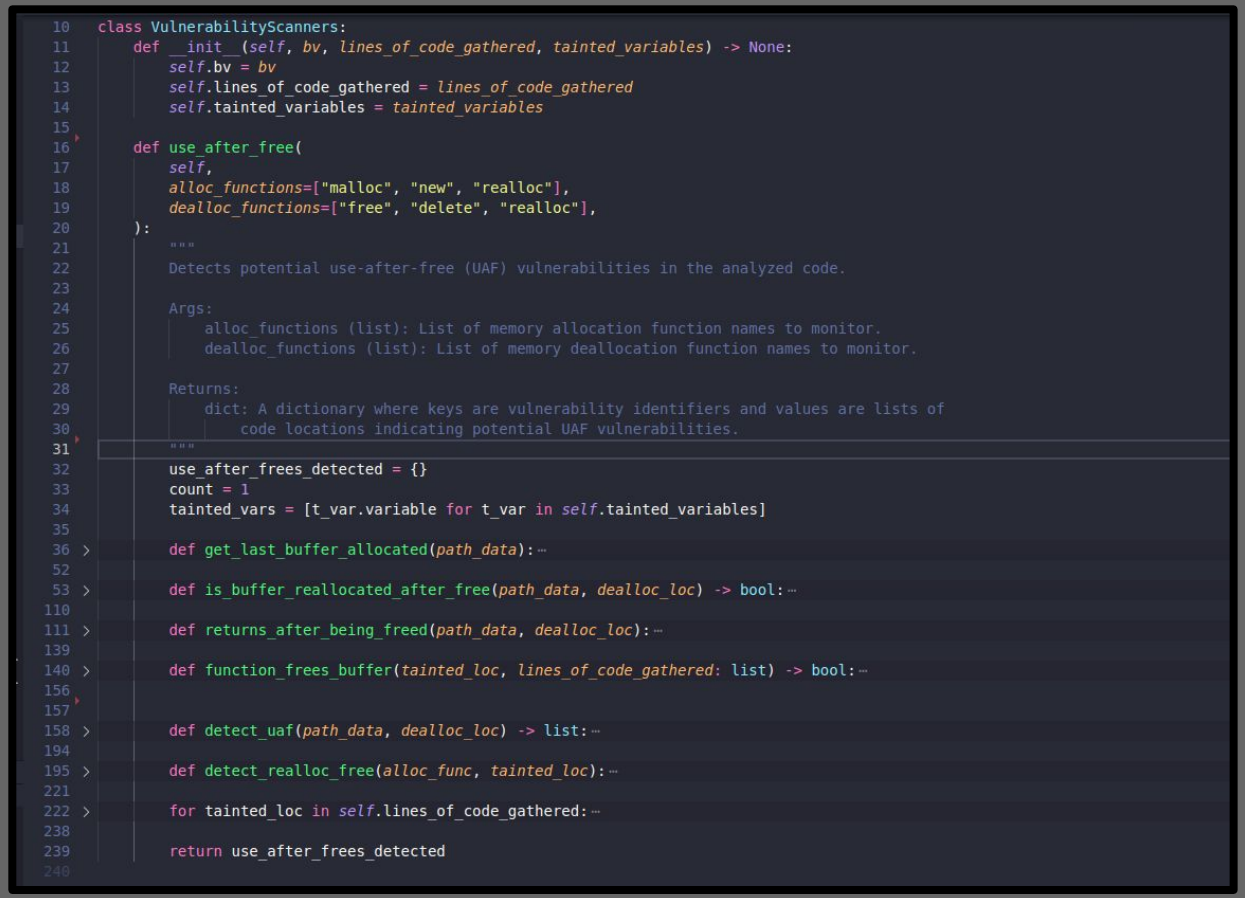


A use-after-free bug occurs when a program continues to access memory after it has been freed.

An attacker can abuse this bug to reallocate freed memory in a controlled way and write arbitrary data to sensitive memory locations under the right conditions.

Use-after-free detection 1/7

Here we create a function to take in analysis output from BinGoggles and determine whether or not a UAF exists in the variable flow path



```python
class VulnerabilityScanners:
    def __init__(self, bv, lines_of_code_gathered, tainted_variables) -> None:
        self.bv = bv
        self.lines_of_code_gathered = lines_of_code_gathered
        self.tainted_variables = tainted_variables

    def use_after_free(
        self,
        alloc_functions=["malloc", "new", "realloc"],
        dealloc_functions=["free", "delete", "realloc"],
    ):
        """
        Detects potential use-after-free (UAF) vulnerabilities in the analyzed code.

        Args:
            alloc_functions (list): List of memory allocation function names to monitor.
            dealloc_functions (list): List of memory deallocation function names to monitor.

        Returns:
            dict: A dictionary where keys are vulnerability identifiers and values are lists of
                code locations indicating potential UAF vulnerabilities.
        """
        use_after_frees_detected = {}
        count = 1
        tainted_vars = [t_var.variable for t_var in self.tainted_variables]

        def get_last_buffer_allocated(path_data):…

        def is_buffer_reallocated_after_free(path_data, dealloc_loc) -> bool:…

        def returns_after_being_freed(path_data, dealloc_loc):…

        def function_frees_buffer(tainted_loc, lines_of_code_gathered: list) -> bool:…


        def detect_uaf(path_data, dealloc_loc) -> list:…

        def detect_realloc_free(alloc_func, tainted_loc):…

        for tainted_loc in self.lines_of_code_gathered:…

        return use_after_frees_detected
```

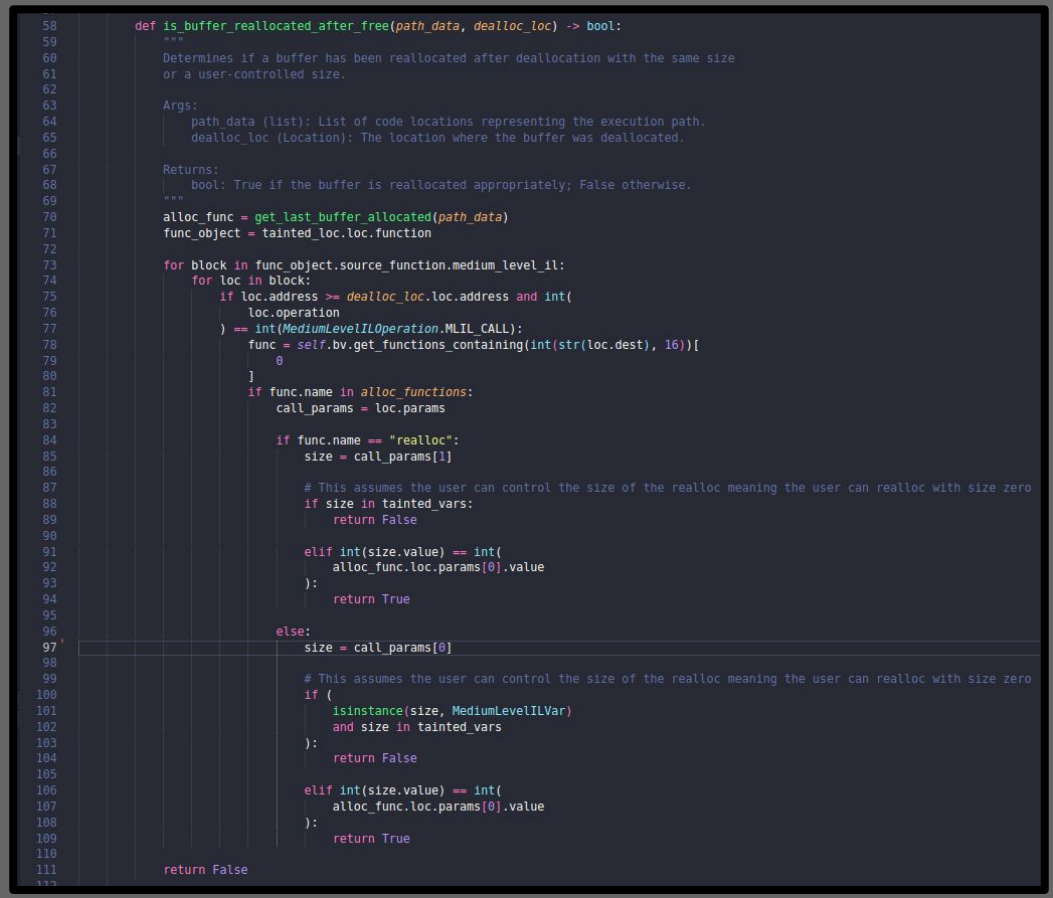
Use-after-free detection 2/7

```
200         def detect_realloc_free(alloc_func, tainted_loc):
201             """
202             Checks if a reallocation function call can lead to a deallocation when the size
203             parameter is zero or user-controlled.
204
205             Args:
206                 alloc_func (Function): The allocation function being analyzed.
207                 tainted_loc (Location): The location of the function call with potential taint.
208
209             Returns:
210                 bool: True if the reallocation can result in deallocation; False otherwise.
211             """
212             if alloc_func.name == "realloc":
213                 call_params = tainted_loc.loc.params
214                 size = call_params[1]
215
216                 if isinstance(size, MediumLevelILVar) and size in tainted_vars:
217                     return True
218
219                 elif (
220                     int(size.operation) == int(MediumLevelILOperation.MLIL_CONST)
221                     and int(size.value) == 0
222                 ):
223                     return True
224
225             return False
```

Use-after-free detection 3/7

```
41          def get_last_buffer_allocated(path_data):
42              for tainted_loc in path_data:
43                  if int(tainted_loc.loc.operation) == int(
44                      MediumLevelILOperation.MLIL_CALL
45                  ):
46                      func = self.bv.get_functions_containing(
47                          int(str(tainted_loc.loc.dest), 16)
48                      )[0]
49                      if func.name in alloc_functions:
50                          # Treat realloc differently
51                          if func.name == "realloc" and not detect_realloc_free(
52                              func, tainted_loc
53                          ):
54                              return tainted_loc
55
56                          return tainted_loc
```

Use-after-free detection 4/7



```python
def is_buffer_reallocated_after_free(path_data, dealloc_loc) -> bool:
    """
    Determines if a buffer has been reallocated after deallocation with the same size
    or a user-controlled size.

    Args:
        path_data (list): List of code locations representing the execution path.
        dealloc_loc (Location): The location where the buffer was deallocated.

    Returns:
        bool: True if the buffer is reallocated appropriately; False otherwise.
    """
    alloc_func = get_last_buffer_allocated(path_data)
    func_object = tainted_loc.loc.function

    for block in func_object.source_function.medium_level_il:
        for loc in block:
            if loc.address >= dealloc_loc.loc.address and int(
                loc.operation
            ) == int(MediumLevelILOperation.MLIL_CALL):
                func = self.bv.get_functions_containing(int(str(loc.dest), 16))[
                    0
                ]
                if func.name in alloc_functions:
                    call_params = loc.params

                    if func.name == "realloc":
                        size = call_params[1]

                        # This assumes the user can control the size of the realloc meaning the user can realloc with size zero
                        if size in tainted_vars:
                            return False

                        elif int(size.value) == int(
                            alloc_func.loc.params[0].value
                        ):
                            return True

                    else:
                        size = call_params[0]

                        # This assumes the user can control the size of the realloc meaning the user can realloc with size zero
                        if (
                            isinstance(size, MediumLevelILVar)
                            and size in tainted_vars
                        ):
                            return False

                        elif int(size.value) == int(
                            alloc_func.loc.params[0].value
                        ):
                            return True

    return False
```

Use-after-free detection 5/7

```python
def returns_after_being_freed(path_data, dealloc_loc):
    """
    Determines if the function returns or jumps immediately after a buffer is freed.

    Args:
        path_data (list): List of code locations representing the execution path.
        dealloc_loc (Location): The location where the buffer was deallocated.

    Returns:
        bool: True if the function returns or jumps immediately after deallocation; False otherwise.
    """
    for tainted_loc in path_data:
        if tainted_loc.loc.address >= dealloc_loc.loc.address and int(
            tainted_loc.loc.operation
        ) == int(MediumLevelILOperation.MLIL_RET):
            return True

        elif tainted_loc.loc.address >= dealloc_loc.loc.address and int(
            tainted_loc.loc.operation
        ) == int(MediumLevelILOperation.MLIL_GOTO):
            func_obj = tainted_loc.loc.function
            jump_addr = tainted_loc.loc.source_location.address
            if int(func_obj.get_llil_at(jump_addr).mlil.operation) == int(
                MediumLevelILOperation.MLIL_RET
            ):
                return True

    return False
```

Use-after-free detection 6/7

```python
def detect_uaf(path_data, dealloc_loc) -> list:
    """
    Identifies the sequence of code locations that lead to a potential use-after-free
    vulnerability after a deallocation.

    Args:
        path_data (list): List of code locations representing the execution path.
        dealloc_loc (Location): The location where the buffer was deallocated.

    Returns:
        list: A list of code locations indicating the vulnerable path, or an empty list
            if no vulnerability is detected.
    """
    vulnerable_path = []

    if is_buffer_reallocated_after_free(path_data, dealloc_loc):
        return []

    elif returns_after_being_freed(path_data, dealloc_loc):
        return []

    for t_loc in path_data:
        if dealloc_loc.loc.address > t_loc.loc.address:
            continue

        if any(var in tainted_vars for var in t_loc.loc.vars_read) or any(
            var in tainted_vars for var in t_loc.loc.vars_written
        ):
            vulnerable_path.append(t_loc)

    return vulnerable_path
```

Use-after-free detection 7/7

```
218         for tainted_loc in self.lines_of_code_gathered:
219             if int(tainted_loc.loc.operation) == int(MediumLevelILOperation.MLIL_CALL):
220                 func = self.bv.get_functions_containing(
221                     int(str(tainted_loc.loc.dest), 16)
222                 )[0]
223 
224                 # Tracing the paths for post dealloction functions
225                 if (
226                     func.name in dealloc_functions
227                     or detect_realloc_free(func, tainted_loc)
228                     or function_frees_buffer(tainted_loc, self.lines_of_code_gathered)
229                 ):
230                     vuln_path = detect_uaf(self.lines_of_code_gathered, tainted_loc)
231                     if len(vuln_path) > 0:
232                         use_after_frees_detected[count] = vuln_path
233                         count += 1
234 
235         return use_after_frees_detected
236 
```

Now it’s time to show off what we can do

```python
class VulnerabilityScanners:
    def __init__(self, bv, lines_of_code_gathered, tainted_variables) -> None:
        self.bv = bv
        self.lines_of_code_gathered = lines_of_code_gathered
        self.tainted_variables = tainted_variables

    def use_after_free(
        self,
        alloc_functions=["malloc", "new", "realloc"],
        dealloc_functions=["free", "delete", "realloc"],
    ):
        """
        Detects potential use-after-free (UAF) vulnerabilities in the analyzed code.

        Args:
            alloc_functions (list): List of memory allocation function names to monitor.
            dealloc_functions (list): List of memory deallocation function names to monitor.

        Returns:
            dict: A dictionary where keys are vulnerability identifiers and values are lists of
                code locations indicating potential UAF vulnerabilities.
        """
        use_after_frees_detected = {}
        count = 1
        tainted_vars = [t_var.variable for t_var in self.tainted_variables]

        def get_last_buffer_allocated(path_data):…

        def is_buffer_reallocated_after_free(path_data, dealloc_loc) -> bool:…

        def returns_after_being_freed(path_data, dealloc_loc):…

        def function_frees_buffer(tainted_loc, lines_of_code_gathered: list) -> bool:…


        def detect_uaf(path_data, dealloc_loc) -> list:…

        def detect_realloc_free(alloc_func, tainted_loc):…

        for tainted_loc in self.lines_of_code_gathered:…

        return use_after_frees_detected
```

VS

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0040125a  buf = malloc(bytes: 0x64)   [PASS]

        Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed
        Level Two:   (VULNERABLE):     0 @ 00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 004013c7  buf = malloc(bytes: 0x64)   [PASS]

        Level Four: Testing UAF where memory is freed and then accessed across function boundaries.
        el Four:  (VULNERABLE):     0 @ 004014e5  buf = malloc(bytes: 0x64)   [PASS]

        Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=True, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x0040125a, "buf"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    print(data)
```

Testing a basic UAF where memory is allocated, freed, and then accessed.

Level 1 Source code

```
5  void level_one() {
6      char *user_input;
7
8      user_input = (char *)malloc(100 * sizeof(char));
9      if (user_input == NULL) {
10         perror("malloc failed");
11         return;
12     }
13
14     printf("Enter a string: ");
15     fgets(user_input, 100, stdin);
16
17     free(user_input);
18     printf("Memory has been freed.\n");
19
20     fgets(user_input, 100, stdin); // UAF occurs here
21     printf("Content after writing: %s\n", user_input); // UAF
22 }
```

Level 1 MLIL

```
00401249    int64_t level_one()

 0 @ 0040125a  buf = malloc(bytes: 0x64)
 1 @ 0040125f  user_input = buf
 2 @ 00401268  if (user_input != 0) then 3 @ 0x401285 else 22 @ 0x401274

 3 @ 00401285  rax_1 = 0
 4 @ 0040128a  printf(format: "Enter a string: ")
 5 @ 0040128f  rdx_1 = [&__TMC_END__]
 6 @ 00401296  rax_2 = user_input
 7 @ 0040129f  rdi_1 = rax_2
 8 @ 004012a2  fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
 9 @ 004012a7  rax_3 = user_input
10 @ 004012ab  rdi_2 = rax_3
11 @ 004012ae  free(mem: rdi_2)
12 @ 004012bd  puts(str: "Memory has been freed.")
13 @ 004012c2  rdx_2 = [&__TMC_END__]
14 @ 004012c9  rax_4 = user_input
15 @ 004012d2  rdi_3 = rax_4
16 @ 004012d5  fgets(buf: rdi_3, n: 0x64, fp: rdx_2)
17 @ 004012da  rax_5 = user_input
18 @ 004012de  rsi_1 = rax_5
19 @ 004012eb  rax_6 = 0
20 @ 004012f0  rax = printf(format: "Content after writing: %s\n", rsi_1)
21 @ 004012f0  goto 24 @ 0x4012f6

22 @ 00401274  rax = perror(s: "malloc failed")
23 @ 00401279  goto 24 @ 0x4012f6

24 @ 004012f6  return rax
```

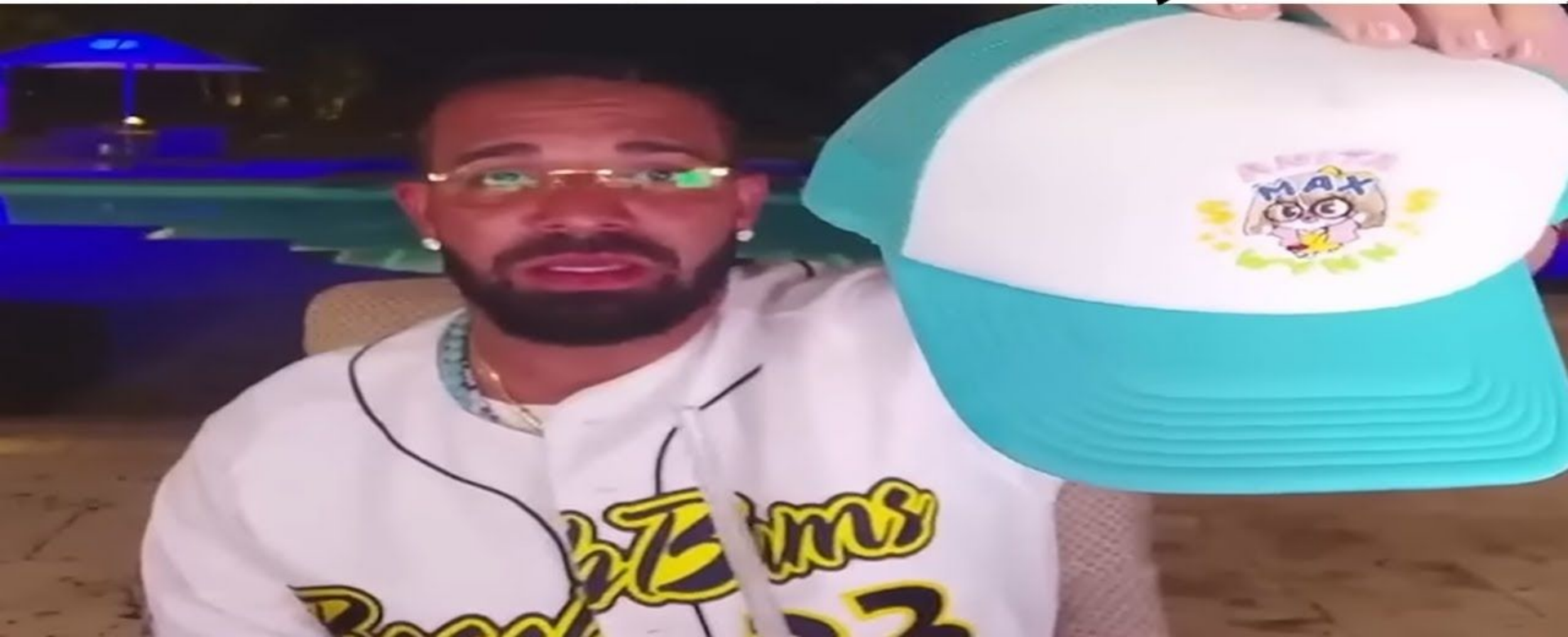
Level 1 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0x0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memor
        Level Two:   (VULNERABLE):     0 @ 0x00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 0x004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 0x004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 0x004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x0040125a, "buf"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 1 Vulnerability detection

```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_uaf
============================================================================================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Loading binary...    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
{1: [[0x4012ae] free(rdi_2) -> rdi_2 -> rax_3 [Tainted],
     [0x4012c9] rax_4 = user_input -> user_input -> buf [Tainted],
     [0x4012d2] rdi_3 = rax_4 -> rax_4 -> user_input [Tainted],
     [0x4012d5] fgets(rdi_3, 0x64, rdx_2) -> rdi_3 -> rax_4 [Tainted],
     [0x4012da] rax_5 = user_input -> user_input -> buf [Tainted],
     [0x4012de] rsi_1 = rax_5 -> rax_5 -> user_input [Tainted],
     [0x4012f0] rax = printf("Content after writing: %s\n", rsi_1) -> rsi_1 -> rax_5 [Tainted]]}
.
```

Anita Max Wynn



Level 2 Source code

Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memory.

```c
void level_two() {
    char *buffer = (char *)malloc(100 * sizeof(char));

    if (buffer == NULL) {
        perror("malloc failed");
        return;
    }

    printf("Enter a string for level_two: ");
    fgets(buffer, 100, stdin);

    // Using realloc to "free" the buffer. This is the key UAF demonstration.
    // realloc with size 0 is equivalent to freeing the memory.
    buffer = (char *)realloc(buffer, 0);  // realloc to free the buffer (buffer is now invalid)

    // After realloc, the buffer is invalid, but we still use it.
    printf("Buffer after realloc (freed memory): ");
    fgets(buffer, 100, stdin); // UAF occurs here, buffer is freed but used
    printf("Content after writing: %s\n", buffer); // UAF
}
```

Level 2 MLIL

```
004012f7    int64_t level_two()

  0 @ 00401308  rax = malloc(bytes: 0x64)
  1 @ 0040130d  buffer = rax
  2 @ 00401316  if (buffer != 0) then 3 @ 0x401336 else 24 @ 0x401322

  3 @ 00401336  rax_2 = 0
  4 @ 0040133b  printf(format: "Enter a string for level_two: ")
  5 @ 00401340  rdx_1 = [&__TMC_END__]
  6 @ 00401347  rax_3 = buffer
  7 @ 00401350  rdi_1 = rax_3
  8 @ 00401353  fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
  9 @ 00401358  rax_4 = buffer
 10 @ 00401361  rdi_2 = rax_4
 11 @ 00401364  buf = realloc(oldmem: rdi_2, bytes: 0)
 12 @ 00401369  buffer = buf
 13 @ 00401377  rax_5 = 0
 14 @ 0040137c  printf(format: "Buffer after realloc (freed memo…")
 15 @ 00401381  rdx_2 = [&__TMC_END__]
 16 @ 00401388  rax_6 = buffer
 17 @ 00401391  rdi_3 = rax_6
 18 @ 00401394  fgets(buf: rdi_3, n: 0x64, fp: rdx_2)
 19 @ 00401399  rax_7 = buffer
 20 @ 0040139d  rsi_1 = rax_7
 21 @ 004013aa  rax_8 = 0
 22 @ 004013af  rax_1 = printf(format: "Content after writing: %s\n", rsi_1)
 23 @ 004013af  goto 26 @ 0x4013b5

 24 @ 00401322  rax_1 = perror(s: "malloc failed")
 25 @ 00401327  goto 26 @ 0x4013b5

 26 @ 004013b5  return rax_1
```

Level 2 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0x0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memc
        Level Two:   (VULNERABLE):     0 @ 0x00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 0x004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 0x004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 0x004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x00401308, "rax"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 2 Vulnerability detection

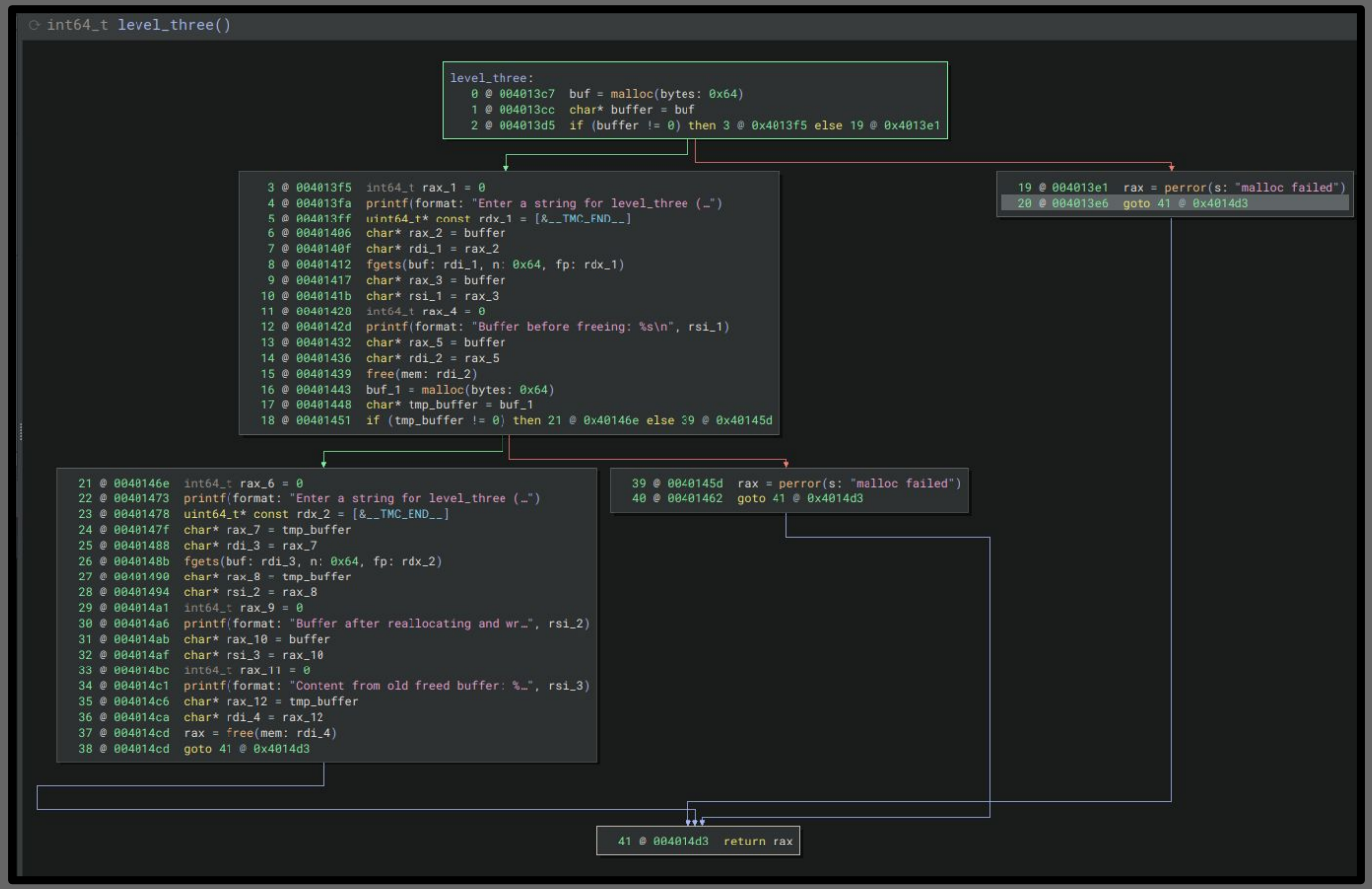
```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_uaf
============================================================================================================ test session starts
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Loading binary...    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
{1: [[0x401364] buf = realloc(rdi_2, 0) -> buf -> rdi_2 [Tainted],
     [0x401369] buffer = buf -> buffer -> buf [Tainted],
     [0x401388] rax_6 = buffer -> buffer -> buf [Tainted],
     [0x401391] rdi_3 = rax_6 -> rax_6 -> buffer [Tainted],
     [0x401394] fgets(rdi_3, 0x64, rdx_2) -> rdi_3 -> rax_6 [Tainted],
     [0x401399] rax_7 = buffer -> buffer -> buf [Tainted],
     [0x40139d] rsi_1 = rax_7 -> rax_7 -> buffer [Tainted],
     [0x4013af] rax_1 = printf("Content after writing: %s\n", rsi_1) -> rsi_1 -> rax_7 [Tainted]]}
```

Level 3 Source code

No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.

```c
void level_three() {
    char *buffer = (char *)malloc(100 * sizeof(char));
    char *tmp_buffer;

    if (buffer == NULL) {
        perror("malloc failed");
        return;
    }

    printf("Enter a string for level_three (before free): ");
    fgets(buffer, 100, stdin);
    printf("Buffer before freeing: %s\n", buffer);

    free(buffer);

    tmp_buffer = (char *)malloc(100 * sizeof(char)); // Reallocates memory
    if (tmp_buffer == NULL) {
        perror("malloc failed");
        return;
    }

    printf("Enter a string for level_three (after free, but before buffer moves): ");
    fgets(tmp_buffer, 100, stdin);

    // After memory has been freed and reallocated, test if old data persists
    printf("Buffer after reallocating and writing new content: %s\n", tmp_buffer);

    // Use the old buffer that was freed
    printf("Content from old freed buffer: %s\n", buffer);
    free(tmp_buffer);
}
```

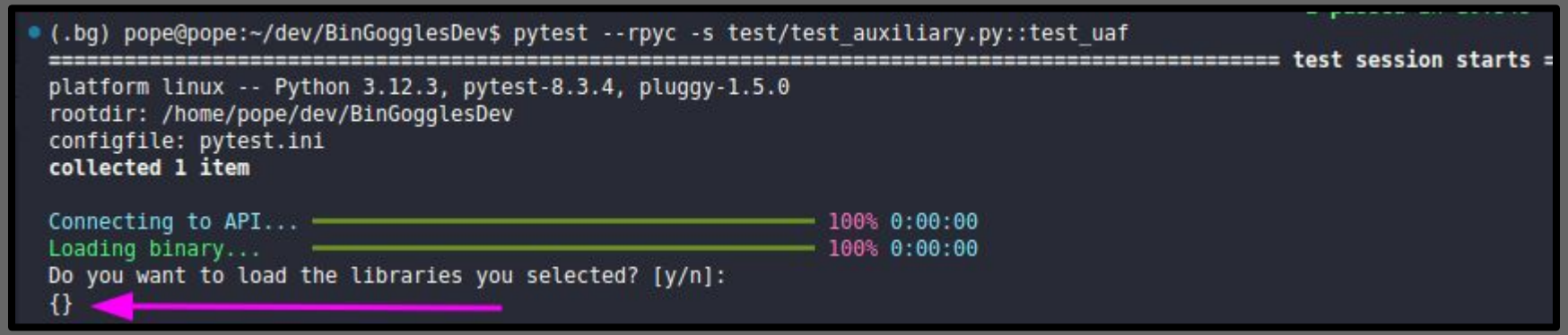
Level 3 MLIL


int64_t level_three()
level_three:
0 @ 004013c7 buf = malloc(bytes: 0x64)
1 @ 004013cc char* buffer = buf
2 @ 004013d5 if (buffer != 0) then 3 @ 0x4013f5 else 19 @ 0x4013e1
3 @ 004013f5 int64_t rax_1 = 0
4 @ 004013fa printf(format: "Enter a string for level_three (…")
5 @ 004013ff uint64_t* const rdx_1 = [&__TMC_END__]
6 @ 00401406 char* rax_2 = buffer
7 @ 0040140f char* rdi_1 = rax_2
8 @ 00401412 fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
9 @ 00401417 char* rax_3 = buffer
10 @ 0040141b char* rsi_1 = rax_3
11 @ 00401428 int64_t rax_4 = 0
12 @ 0040142d printf(format: "Buffer before freeing: %s\n", rsi_1)
13 @ 00401432 char* rax_5 = buffer
14 @ 00401436 char* rdi_2 = rax_5
15 @ 00401439 free(mem: rdi_2)
16 @ 00401443 buf_1 = malloc(bytes: 0x64)
17 @ 00401448 char* tmp_buffer = buf_1
18 @ 00401451 if (tmp_buffer != 0) then 21 @ 0x40146e else 39 @ 0x40145d
19 @ 004013e1 rax = perror(s: "malloc failed")
20 @ 004013e6 goto 41 @ 0x4014d3
21 @ 0040146e int64_t rax_6 = 0
22 @ 00401473 printf(format: "Enter a string for level_three (…")
23 @ 00401478 uint64_t* const rdx_2 = [&__TMC_END__]
24 @ 0040147f char* rax_7 = tmp_buffer
25 @ 00401488 char* rdi_3 = rax_7
26 @ 0040148b fgets(buf: rdi_3, n: 0x64, fp: rdx_2)
27 @ 00401490 char* rax_8 = tmp_buffer
28 @ 00401494 char* rsi_2 = rax_8
29 @ 004014a1 int64_t rax_9 = 0
30 @ 004014a6 printf(format: "Buffer after reallocating and wr…", rsi_2)
31 @ 004014ab char* rax_10 = buffer
32 @ 004014af char* rsi_3 = rax_10
33 @ 004014bc int64_t rax_11 = 0
34 @ 004014c1 printf(format: "Content from old freed buffer: %…", rsi_3)
35 @ 004014c6 char* rax_12 = tmp_buffer
36 @ 004014ca char* rdi_4 = rax_12
37 @ 004014cd rax = free(mem: rdi_4)
38 @ 004014cd goto 41 @ 0x4014d3
39 @ 0040145d rax = perror(s: "malloc failed")
40 @ 00401462 goto 41 @ 0x4014d3
41 @ 004014d3 return rax

Level 3 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memory.
        Level Two:   (VULNERABLE):     0 @ 00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x004013c7, "buf"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 3 Vulnerability detection



Level 4 Source code

Testing UAF detection when the vulnerable ptr is assigned to another ptr after being freed.

```c
void level_four() {
    char *buffer_a = (char *)malloc(100 * sizeof(char));
    char *buffer_b;

    if (buffer_a == NULL) {
        perror("malloc failed");
        return;
    }

    printf("Enter a string for level_four (buffer_a): ");
    fgets(buffer_a, 100, stdin);
    printf("Buffer A before freeing: %s\n", buffer_a);

    free(buffer_a); // Free buffer_a in this function

    buffer_b = buffer_a; // reassign buffer_a after the free

    // We have passed the freed buffer to buffer_b, and now we use it
    printf("Buffer B after passing freed buffer: ");
    fgets(buffer_b, 100, stdin); // UAF occurs here

    printf("Content from buffer_b after UAF: %s\n", buffer_b); // UAF
}
```

Level 4 MLIL

```
004014d4    int64_t level_four()

  0 @ 004014e5  buf = malloc(bytes: 0x64)
  1 @ 004014ea  buffer_a = buf
  2 @ 004014f3  if (buffer_a != 0) then 3 @ 0x401513 else 29 @ 0x4014ff

  3 @ 00401513  rax_1 = 0
  4 @ 00401518  printf(format: "Enter a string for level_four (b…")
  5 @ 0040151d  rdx_1 = [&__TMC_END__]
  6 @ 00401524  rax_2 = buffer_a
  7 @ 0040152d  rdi_1 = rax_2
  8 @ 00401530  fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
  9 @ 00401535  rax_3 = buffer_a
 10 @ 00401539  rsi_1 = rax_3
 11 @ 00401546  rax_4 = 0
 12 @ 0040154b  printf(format: "Buffer A before freeing: %s\n", rsi_1)
 13 @ 00401550  rax_5 = buffer_a
 14 @ 00401554  rdi_2 = rax_5
 15 @ 00401557  free(mem: rdi_2)
 16 @ 0040155c  rax_6 = buffer_a
 17 @ 00401560  buffer_b = rax_6
 18 @ 0040156e  rax_7 = 0
 19 @ 00401573  printf(format: "Buffer B after passing freed buf…")
 20 @ 00401578  rdx_2 = [&__TMC_END__]
 21 @ 0040157f  rax_8 = buffer_b
 22 @ 00401588  rdi_3 = rax_8
 23 @ 0040158b  fgets(buf: rdi_3, n: 0x64, fp: rdx_2)
 24 @ 00401590  rax_9 = buffer_b
 25 @ 00401594  rsi_2 = rax_9
 26 @ 004015a1  rax_10 = 0
 27 @ 004015a6  rax = printf(format: "Content from buffer_b after UAF:…", rsi_2)
 28 @ 004015a6  goto 31 @ 0x4015ac

 29 @ 004014ff  rax = perror(s: "malloc failed")
 30 @ 00401504  goto 31 @ 0x4015ac

 31 @ 004015ac  return rax
```

Level 4 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0x0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed mem
        Level Two:   (VULNERABLE):     0 @ 0x00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 0x004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 0x004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 0x004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x004014e5, "buf"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 4 vulnerability detection

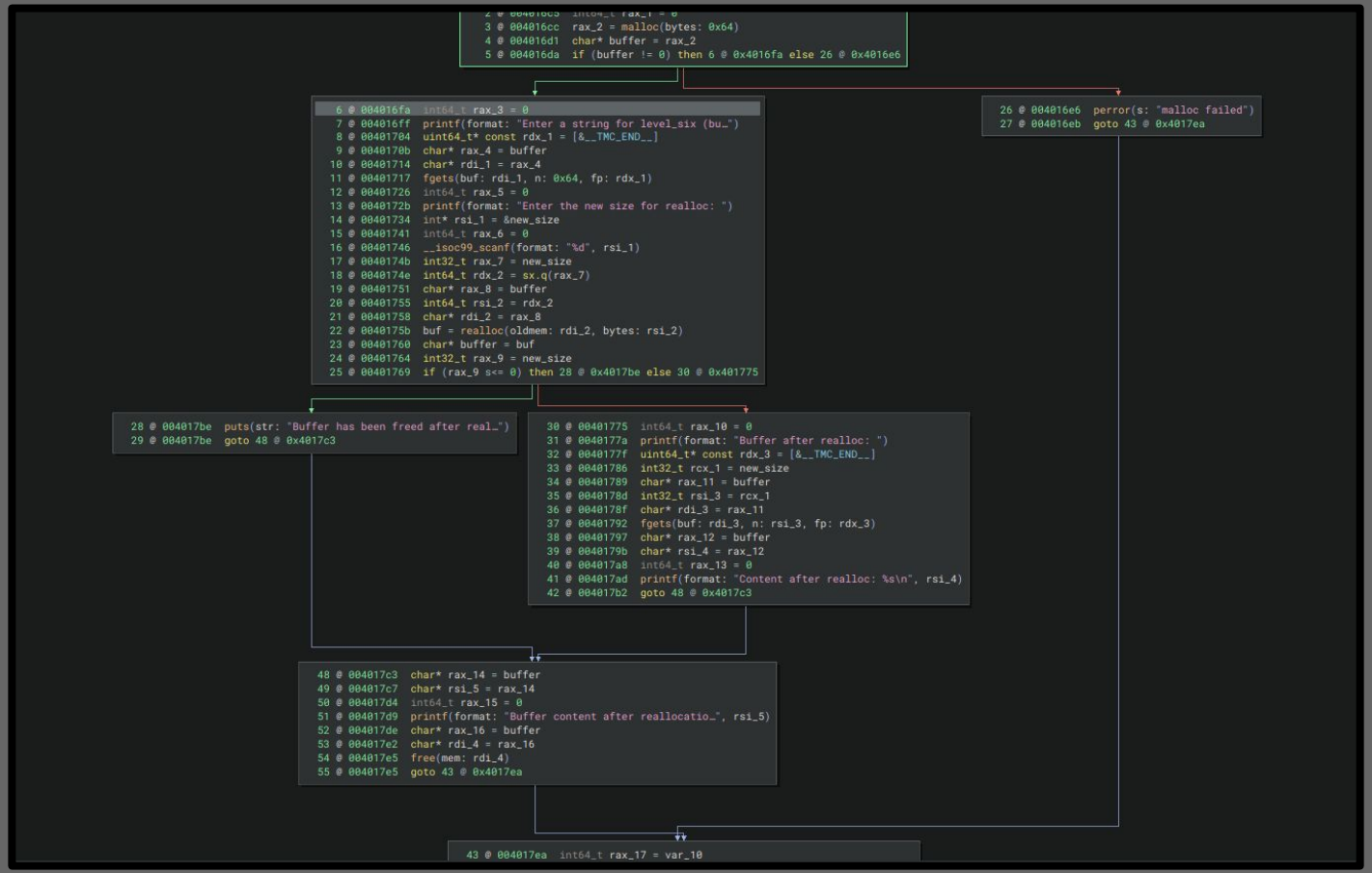
```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_uaf
============================================================================================== test session starts ==============
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ──────────────────────────────────────── 100% 0:00:00
Loading binary...    ──────────────────────────────────────── 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
{1: [[0x401557] free(rdi_2) -> rdi_2 -> rax_5 [Tainted],
     [0x40155c] rax_6 = buffer_a -> buffer_a -> buf [Tainted],
     [0x401560] buffer_b = rax_6 -> rax_6 -> buffer_a [Tainted],
     [0x40157f] rax_8 = buffer_b -> buffer_b -> rax_6 [Tainted],
     [0x401588] rdi_3 = rax_8 -> rax_8 -> buffer_b [Tainted],
     [0x40158b] fgets(rdi_3, 0x64, rdx_2) -> rdi_3 -> rax_8 [Tainted],
     [0x401590] rax_9 = buffer_b -> buffer_b -> rax_6 [Tainted],
     [0x401594] rsi_2 = rax_9 -> rax_9 -> buffer_b [Tainted],
     [0x4015a6] rax = printf("Content from buffer_b after UAF:…", rsi_2) -> rsi_2 -> rax_9 [Tainted]]}
.
```

Level 5 Source code

UAF where memory is reallocated but used after being freed by realloc

```c
void level_five() {
    char *buffer = (char *)malloc(100 * sizeof(char));
    int new_size;

    if (buffer == NULL) {
        perror("malloc failed");
        return;
    }

    printf("Enter a string for level_six (buffer): ");
    fgets(buffer, 100, stdin);

    // Ask the user for the new size to realloc
    printf("Enter the new size for realloc: ");
    scanf("%d", &new_size);

    // Reallocate memory with user-controlled size
    buffer = (char *)realloc(buffer, new_size);  // User controls the size here

    // After realloc, we attempt to use the buffer again, potentially causing a vulnerability
    if (new_size > 0) {
        printf("Buffer after realloc: ");
        fgets(buffer, new_size, stdin); // Using the reallocated buffer
        printf("Content after realloc: %s\n", buffer);
    } else {
        // If realloc size is 0, the buffer is freed
        printf("Buffer has been freed after realloc with size 0.\n");
    }

    // We still use the buffer (it could be freed if realloc size is 0)
    printf("Buffer content after reallocation: %s\n", buffer);
    free(buffer);
}
```

Level 5 MLIL


3 @ 004016cc rax_2 = malloc(bytes: 0x64)
4 @ 004016d1 char* buffer = rax_2
5 @ 004016da if (buffer != 0) then 6 @ 0x4016fa else 26 @ 0x4016e6
6 @ 004016fa int64_t rax_3 = 0
7 @ 004016ff printf(format: "Enter a string for level_six (bu…")
8 @ 00401704 uint64_t* const rdx_1 = [&__TMC_END__]
9 @ 0040170b char* rax_4 = buffer
10 @ 00401714 char* rdi_1 = rax_4
11 @ 00401717 fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
12 @ 00401726 int64_t rax_5 = 0
13 @ 0040172b printf(format: "Enter the new size for realloc: ")
14 @ 00401734 int* rsi_1 = &new_size
15 @ 00401741 int64_t rax_6 = 0
16 @ 00401746 __isoc99_scanf(format: "%d", rsi_1)
17 @ 0040174b int32_t rax_7 = new_size
18 @ 0040174e int64_t rdx_2 = sx.q(rax_7)
19 @ 00401751 char* rax_8 = buffer
20 @ 00401755 int64_t rsi_2 = rdx_2
21 @ 00401758 char* rdi_2 = rax_8
22 @ 0040175b buf = realloc(oldmem: rdi_2, bytes: rsi_2)
23 @ 00401760 char* buffer = buf
24 @ 00401764 int32_t rax_9 = new_size
25 @ 00401769 if (rax_9 s<= 0) then 28 @ 0x4017be else 30 @ 0x401775
26 @ 004016e6 perror(s: "malloc failed")
27 @ 004016eb goto 43 @ 0x4017ea
28 @ 004017be puts(str: "Buffer has been freed after real…")
29 @ 004017be goto 48 @ 0x4017c3
30 @ 00401775 int64_t rax_10 = 0
31 @ 0040177a printf(format: "Buffer after realloc: ")
32 @ 0040177f uint64_t* const rdx_3 = [&__TMC_END__]
33 @ 00401786 int32_t rcx_1 = new_size
34 @ 00401789 char* rax_11 = buffer
35 @ 0040178d int32_t rsi_3 = rcx_1
36 @ 0040178f char* rdi_3 = rax_11
37 @ 00401792 fgets(buf: rdi_3, n: rsi_3, fp: rdx_3)
38 @ 00401797 char* rax_12 = buffer
39 @ 0040179b char* rsi_4 = rax_12
40 @ 004017a8 int64_t rax_13 = 0
41 @ 004017ad printf(format: "Content after realloc: %s\n", rsi_4)
42 @ 004017b2 goto 48 @ 0x4017c3
48 @ 004017c3 char* rax_14 = buffer
49 @ 004017c7 char* rsi_5 = rax_14
50 @ 004017d4 int64_t rax_15 = 0
51 @ 004017d9 printf(format: "Buffer content after reallocatio…", rsi_5)
52 @ 004017de char* rax_16 = buffer
53 @ 004017e2 char* rdi_4 = rax_16
54 @ 004017e5 free(mem: rdi_4)
55 @ 004017e5 goto 43 @ 0x4017ea
43 @ 004017ea int64_t rax_17 = var_10

Level 5 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0x0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memory.
        Level Two:   (VULNERABLE):     0 @ 0x00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 0x004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 0x004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 0x004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x004016cc, "rax_2"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 5 Vulnerability detection

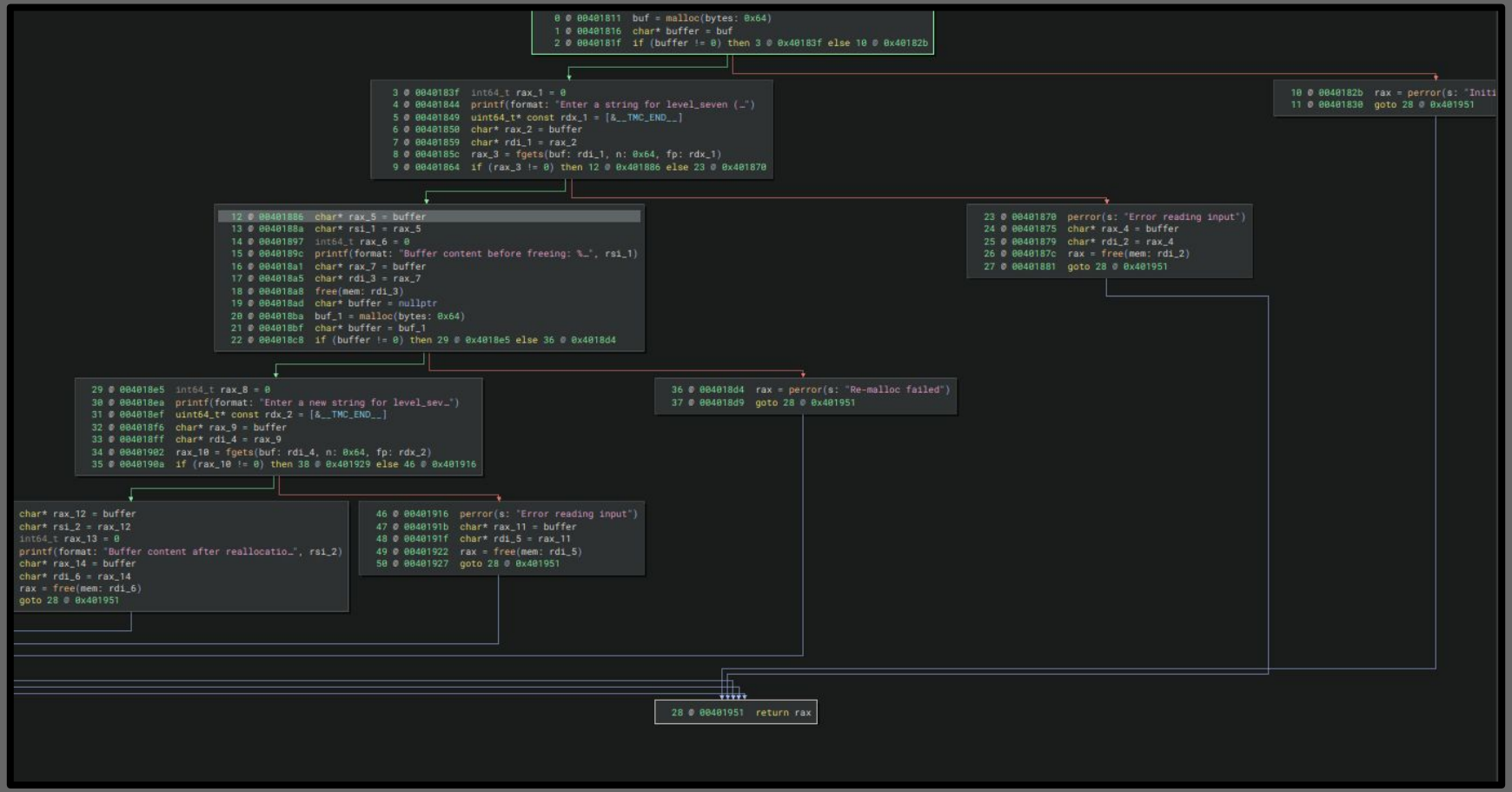
```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_uaf
============================================================================================== test session sta
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ──────────────────────────────────────── 100% 0:00:00
Connecting to API... ──────────────────────────────────────── 100% 0:00:00
Loading binary...    ──────────────────────────────────────── 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
{1: [[0x40175b] buf = realloc(rdi_2, rsi_2) -> buf -> rdi_2 [Tainted],
     [0x401760] buffer = buf -> buffer -> buf [Tainted],
     [0x401789] rax_11 = buffer -> buffer -> buf [Tainted],
     [0x40178f] rdi_3 = rax_11 -> rax_11 -> buffer [Tainted],
     [0x401792] fgets(rdi_3, rsi_3, rdx_3) -> rdi_3 -> rax_11 [Tainted],
     [0x401797] rax_12 = buffer -> buffer -> buf [Tainted],
     [0x40179b] rsi_4 = rax_12 -> rax_12 -> buffer [Tainted],
     [0x4017ad] printf("Content after realloc: %s\n", rsi_4) -> rsi_4 -> rax_12 [Tainted],
     [0x4017c3] rax_14 = buffer -> buffer -> buf [Tainted],
     [0x4017c7] rsi_5 = rax_14 -> rax_14 -> buffer [Tainted],
     [0x4017d9] printf("Buffer content after reallocatio…", rsi_5) -> rsi_5 -> rax_14 [Tainted],
     [0x4017de] rax_16 = buffer -> buffer -> buf [Tainted],
     [0x4017e2] rdi_4 = rax_16 -> rax_16 -> buffer [Tainted],
     [0x4017e5] free(rdi_4) -> rdi_4 -> rax_16 [Tainted]],
 2: [[0x4017e5] free(rdi_4) -> rdi_4 -> rax_16 [Tainted]]}
.
```

Level 6 Source code

Safe usage of memory where allocated memory is correctly freed and reallocated.

```c
void level_six() {
    char *buffer;

    // Step 1: Allocate memory
    buffer = (char *)malloc(100 * sizeof(char));
    if (buffer == NULL) {
        perror("Initial malloc failed");
        return;
    }

    // Step 2: Use the buffer
    printf("Enter a string for level_seven (initial buffer): ");
    if (fgets(buffer, 100, stdin) == NULL) {
        perror("Error reading input");
        free(buffer);
        return;
    }
    printf("Buffer content before freeing: %s\n", buffer);

    // Step 3: Free the buffer
    free(buffer);
    buffer = NULL;

    // Step 4: Reallocate the buffer
    buffer = (char *)malloc(100 * sizeof(char));
    if (buffer == NULL) {
        perror("Re-malloc failed");
        return;
    }

    // Step 5: Use the reallocated buffer
    printf("Enter a new string for level_seven (reallocated buffer): ");
    if (fgets(buffer, 100, stdin) == NULL) {
        perror("Error reading input");
        free(buffer);
        return;
    }
    printf("Buffer content after reallocation: %s\n", buffer);

    // Step 6: Clean up
    free(buffer);
}
```

Level 6 MLIL


0 @ 00401811 buf = malloc(bytes: 0x64)
1 @ 00401816 char* buffer = buf
2 @ 0040181f if (buffer != 0) then 3 @ 0x40183f else 10 @ 0x40182b
3 @ 0040183f int64_t rax_1 = 0
4 @ 00401844 printf(format: "Enter a string for level_seven (_")
5 @ 00401849 uint64_t* const rdx_1 = [&__TMC_END__]
6 @ 00401850 char* rax_2 = buffer
7 @ 00401859 char* rdi_1 = rax_2
8 @ 0040185c rax_3 = fgets(buf: rdi_1, n: 0x64, fp: rdx_1)
9 @ 00401864 if (rax_3 != 0) then 12 @ 0x401886 else 23 @ 0x401870
10 @ 0040182b rax = perror(s: "Initi
11 @ 00401830 goto 28 @ 0x401951
12 @ 00401886 char* rax_5 = buffer
13 @ 0040188a char* rsi_1 = rax_5
14 @ 00401897 int64_t rax_6 = 0
15 @ 0040189c printf(format: "Buffer content before freeing: %_", rsi_1)
16 @ 004018a1 char* rax_7 = buffer
17 @ 004018a5 char* rdi_3 = rax_7
18 @ 004018a8 free(mem: rdi_3)
19 @ 004018ad char* buffer = nullptr
20 @ 004018ba buf_1 = malloc(bytes: 0x64)
21 @ 004018bf char* buffer = buf_1
22 @ 004018c8 if (buffer != 0) then 29 @ 0x4018e5 else 36 @ 0x4018d4
23 @ 00401870 perror(s: "Error reading input")
24 @ 00401875 char* rax_4 = buffer
25 @ 00401879 char* rdi_2 = rax_4
26 @ 0040187c rax = free(mem: rdi_2)
27 @ 00401881 goto 28 @ 0x401951
29 @ 004018e5 int64_t rax_8 = 0
30 @ 004018ea printf(format: "Enter a new string for level_sev_")
31 @ 004018ef uint64_t* const rdx_2 = [&__TMC_END__]
32 @ 004018f6 char* rax_9 = buffer
33 @ 004018ff char* rdi_4 = rax_9
34 @ 00401902 rax_10 = fgets(buf: rdi_4, n: 0x64, fp: rdx_2)
35 @ 0040190a if (rax_10 != 0) then 38 @ 0x401929 else 46 @ 0x401916
36 @ 004018d4 rax = perror(s: "Re-malloc failed")
37 @ 004018d9 goto 28 @ 0x401951
char* rax_12 = buffer
char* rsi_2 = rax_12
int64_t rax_13 = 0
printf(format: "Buffer content after reallocatio_", rsi_2)
char* rax_14 = buffer
char* rdi_6 = rax_14
rax = free(mem: rdi_6)
goto 28 @ 0x401951
46 @ 00401916 perror(s: "Error reading input")
47 @ 0040191b char* rax_11 = buffer
48 @ 0040191f char* rdi_5 = rax_11
49 @ 00401922 rax = free(mem: rdi_5)
50 @ 00401927 goto 28 @ 0x401951
28 @ 00401951 return rax

Level 6 BinGoggles + vulnerability detection

```python
def test_uaf(bg_init, test_bin="./test/binaries/bin/test_uaf"):
    bg = bg_init(
        target_bin=abspath(test_bin),
        libraries=["/lib/x86_64-linux-gnu/libc.so.6"],
        host="127.0.0.1",
        port=18812,
    )
    bv, libraries_mapped = bg.init()

    """
        # Level One: Testing a basic Use-After-Free (UAF) where memory is allocated, freed, and then accessed.
        Level One:   (VULNERABLE):     0 @ 0x0040125a  buf = malloc(bytes: 0x64)   [PASS]

        # Level Two: Testing a UAF using realloc with size 0 (effectively freeing the memory), then accessing the freed memory.
        Level Two:   (VULNERABLE):     0 @ 0x00401308  rax = malloc(bytes: 0x64)   [PASS]

        # Level Three: No vulnerability, testing for safe usage of allocated memory without freeing it prematurely.
        Level Three: (SAFE):           0 @ 0x004013c7  buf = malloc(bytes: 0x64)   [PASS]

        # Level Four: Testing UAF detection when the vulnerable ptr is reassigned to another.
        Level Four:  (VULNERABLE):     0 @ 0x004014e5  buf = malloc(bytes: 0x64)   [PASS]

        # Level Five: UAF where memory is reallocated but used after being freed by realloc.
        Level Five    (VULNERABLE):     3 @ 0x004016cc  rax_2 = malloc(bytes: 0x64) [PASS]

        # Level Six: Safe usage of memory where allocated memory is correctly freed and reallocated.
        Level Six  (SAFE):           0 @ 0x00401811  buf = malloc(bytes: 0x64)   [PASS]
    """

    aux = Analysis(binaryview=bv, verbose=False, libraries_mapped=libraries_mapped)

    locs, _, tainted_vars = aux.tainted_forward_slice(
        target=TaintTarget(0x00401811, "buf"),
        var_type=SlicingID.FunctionVar,
    )

    scanners = VulnerabilityScanners(bv, locs, tainted_vars)
    data = scanners.use_after_free()
    pprint(data)
```

Level 6 Vulnerability detection

```
(.bg) pope@pope:~/dev/BinGogglesDev$ pytest --rpyc -s test/test_auxiliary.py::test_uaf
============================================================================================================ t
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pope/dev/BinGogglesDev
configfile: pytest.ini
collected 1 item

Connecting to API... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Loading binary...    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
Do you want to load the libraries you selected? [y/n]:
{}
.
```

Thank you all for coming I hope that you guys enjoyed the presentation and learned something!

If you have any questions feel free to ask now or hit me through any of my socials and I will try to get back to you.