PENTERA | Pentera Labs™ Research Series

LOLBins Against the Machine

Reverse Engineering at Machine Speed

Matan Abutbul

Table of contents

03 Purpose
03 Executive Summary
04 The Cloak of Legitimacy
05 From a Simple Task to a Bigger Question
06 Old-School LOLBins Hunting
09 Moving to Automation
10 Automating the Manual Process
11 Scaling the Analysis
13 The Bigger Takeaway
13 Summary
13 About Pentera

Purpose

Attackers can utilize Living Off the Land Binaries (LOLBins) to execute commands, evade detection, and maintain persistence using legitimate system tools already present in your environment. This research explores how AI can be used to proactively discover new, undocumented LOLBins before they are weaponized.

Executive Summary

The next LOLBin isn't in a threat intel feed; it's hiding in your `/usr/bin` directory right now. But most defenders only discover it after it's been weaponized. I aim to flip this, helping defenders proactively discover unknown binaries that can be abused for command execution. My research introduces a novel approach that leverages AI to enhance reverse engineering by automating the tracing of execution paths and identifying attacker-relevant functionality. LOLBins serve as a case study for this methodology, but the framework is applicable to a wide range of binary analysis problems that extend far beyond this single category.

Does it apply to my organization?

Yes. If your organization runs Linux, Windows, or macOS environments and relies on built-in system utilities for administration or automation. Even if you have robust endpoint detection, these binaries can be abused for stealthy attacks that blend in with legitimate operations

Who should read this?

Security researchers, red teamers, and defenders (including CISOs and SOC analysts) who want to understand and detect emerging "living off the land" techniques and learn how AI can be used to scale binary analysis and uncover new abuse paths.

The Cloak of Legitimacy

Picture a secure compound. Every vehicle approaching the loading dock is inspected. Unknown cars are stopped, IDs are checked, and questions are asked. But what if, instead of showing up in your own car, you arrive in the delivery truck of a well-known/pre-approved vendor? You are wearing the uniform, driving the right route, and following the expected process. To security, you look like part of the daily routine. No one questions you. You are waved through without a second glance.

This is exactly how attackers operate when they abuse LOLBins. These are legitimate system executables, already present on most machines and used daily for administration. They are signed, trusted, and pre-installed. Because of that, they rarely raise alarms. From the defender's perspective, they look like the vendor truck; part of business as usual.

But attackers see them for what they really are: low-noise, high-leverage tools for stealth and persistence. Instead of dropping custom malware or noisy payloads, they turn the environment's own tools against it. With LOLBins, they can run arbitrary commands, pivot between machines, establish footholds, and exfiltrate data without uploading a single foreign binary or tripping traditional security controls. These actions are wrapped in the cloak of legitimacy, hiding malicious behavior behind the trusted façade of routine operations.

That is what makes LOLBins so effective. Most detection logic is built to flag anomalies; new binaries, unexpected behaviors, or unusual traffic patterns. But LOLBins do not stand out. Their presence is expected, their behavior appears ordinary, and their abuse often mimics real administrative activity. As a result, defenders end up scanning for threats while the attacker is already inside, quietly blending into the background noise of everyday operations.

This write-up focuses on discovering new LOLBins in Linux environments, though the same principle applies across platforms. Windows, macOS, and cloud environments all contain their own versions of native utilities that can be misused in similar ways. The central idea is universal: the more trusted the tool, the more dangerous it becomes when used against you.

From a Simple Task to a Bigger Question

It all started when I was working on a new feature for our product. The feature was intended to add more options for executing commands with elevated privileges. So I went to the well-known repository GTFOBins in order to search for matching binaries for the task.

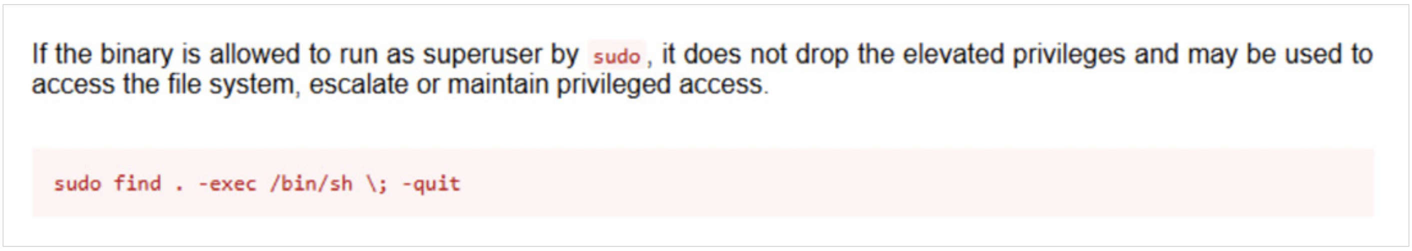
During my work, there was one thought I kept circling back to: How hard is it to create such a repository, and maintain it?
After finishing my initially assigned task, I had time for a little research that combined "old-fashioned" reverse engineering and AI.

The research question was **"Is it possible to use AI in order to automate the process of finding new LOLBins?"** Not just to document them after the fact, but to proactively surface binaries with attacker-useful execution behavior before they are widely known.

The answer is YES, of course, but how?

As this was the leading question I had another thing in mind: I want to use AI to improve my efficiency doing so. I remember back in the days the time and effort I used to spend in order to analyze just one binary.

Before you can effectively automate a process, it's better to understand what the process looks like manually. So I started some old-fashioned reverse engineering of one binary that is already known for privilege escalation (`/usr/bin/find`).


If the binary is allowed to run as superuser by `sudo`, it does not drop the elevated privileges and may be used to access the file system, escalate or maintain privileged access.

```
sudo find . -exec /bin/sh \; -quit
```

Source - https://gtfobins.github.io/gtfobins/find/

What I had in mind was that when using a LOLBin to execute a command or execute another binary, there must be some sort of relevant syscalls that are being used in the process. I wanted to search for those functions and then take their arguments and back-track those arguments to validate if they were passed from the main function using command line arguments.

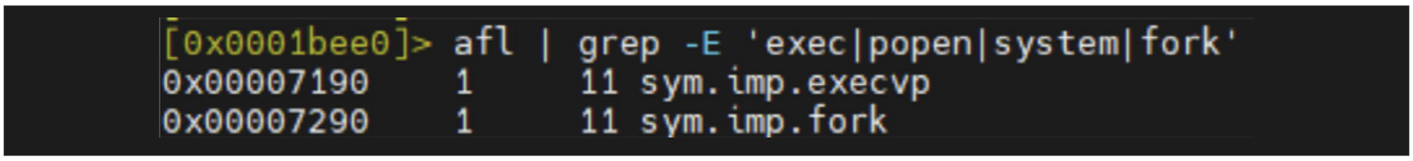
That manual workflow became the foundation for everything that followed.

Old-School LOLBins Hunting

Whenever I plan to automate something, I start by mastering it manually. Before writing a single line of code, I want to understand exactly how the behavior works, what it depends on, and which decisions I'll eventually need to replicate programmatically. In this section, I'll walk through the manual analysis flow I performed using the radare2 framework to determine whether a given binary exhibits LOLBin-like behavior.

The goal of this section is not to teach radare2, but to demonstrate the reasoning process the automation must later replicate.

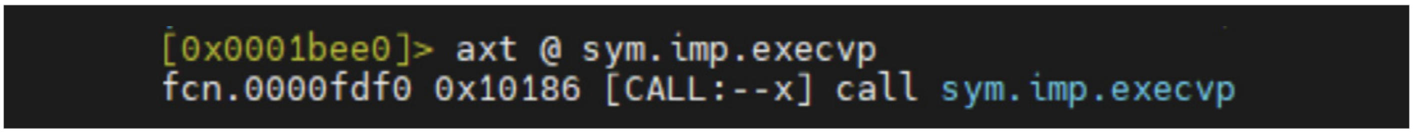
After loading the binary into radare2, I begin by searching for common execution primitives:



```
[0x0001bee0]> afl | grep -E 'exec|popen|system|fork'
0x00007190    1      11 sym.imp.execvp
0x00007290    1      11 sym.imp.fork
```

I start with the first match. Ideally it's the one we're after. If not, I simply continue through the list and repeat the process.

Next, I look for all cross-references to `sym.imp.execvp` effectively asking "Where in the binary is **execvp** actually invoked?":



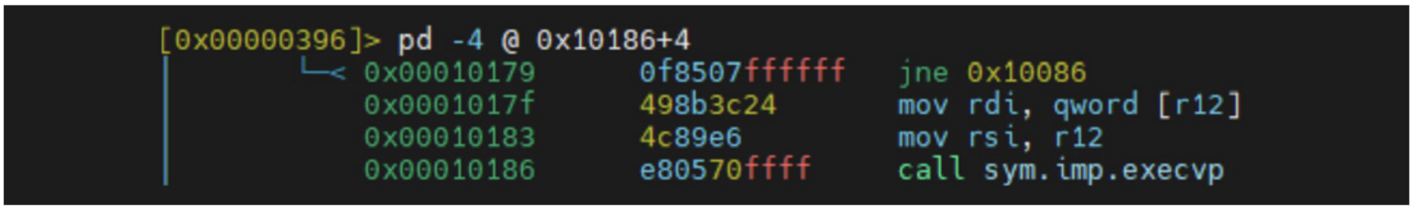
```
[0x0001bee0]> axt @ sym.imp.execvp
fcn.0000fdf0 0x10186 [CALL:--x] call sym.imp.execvp
```

In this case, there is only one caller:
`sym.imp.execvp` is invoked exclusively from `fcn.0000fdf0` at address `0x10186`.
Any external command execution in this binary must pass through this function.

After locating `sym.imp.execvp` and listing its cross-references, I now know exactly which functions in the binary are responsible for spawning external commands. In this sample, **execvp** is called from a single function, `fcn.0000fdf0` at address `0x10186`, so any command execution must flow through this code path.

A bit of information about the function using `afi @ fcn.0000fdf0` : there are 298 instructions, meaning we need to focus on specific blocks. First, we would like to see the parameters of **execvp** and where they come from `pdf @ fcn.0000fdf0` will print the disassembled function where the execvp was called using `pd -4 @ 0x10186+4` (and repeating the same process for every call site when there is more than one).

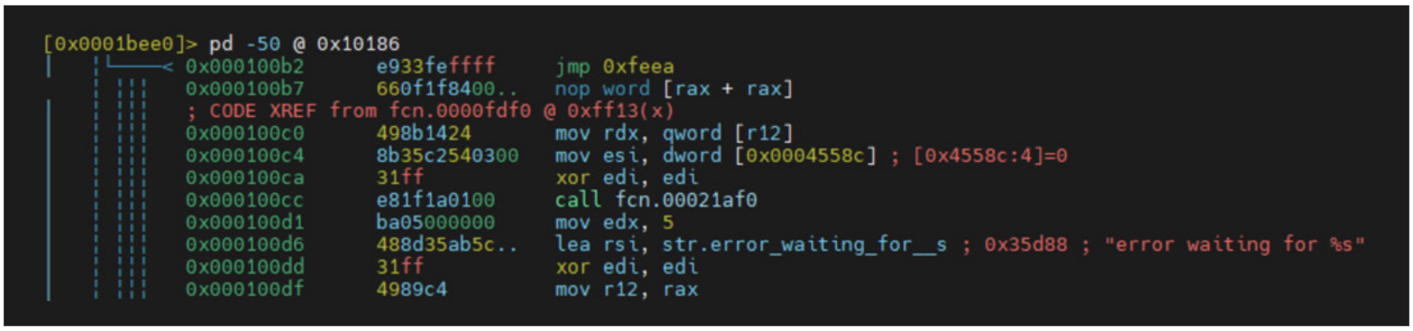
I inspect how the argument registers are populated just before the **execvp** call (4 instructions are enough this time). In this case, the first argument (the program name) is built from `r12`.



```
[0x00000396]> pd -4 @ 0x10186
│       └─< 0x00010179      0f8507ffffff   jne 0x10086
│           0x0001017f      498b3c24       mov rdi, qword [r12]
│           0x00010183      4c89e6         mov rsi, r12
│           0x00010186      e80570ffff     call sym.imp.execvp
```

`r12` is the suspect, this is the variable that holds the binary to execute next.

Around the **execvp** call there are several function calls: `fcn.00021af0, sym.imp.dcgettext, sym.imp.error, fcn.00021cb0`, and `fcn.0001c150`.
To decide where to dig next, we focus on data flow, not just proximity.
execvp uses `r12` as argv, so we look for where `r12` is last defined.

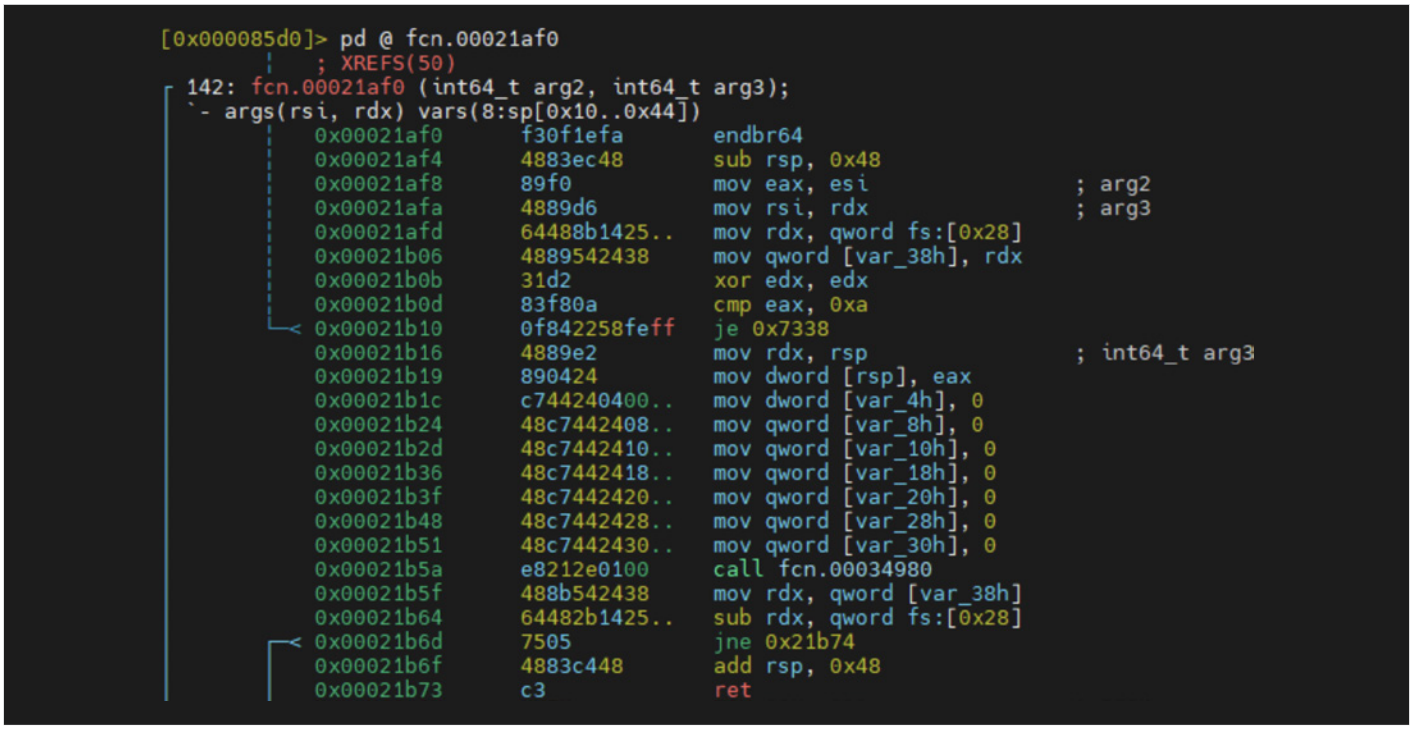


```
[0x0001bee0]> pd -50 @ 0x10186
│   ¦└────< 0x000100b2      e933feffff     jmp 0xfeea
    ¦ ¦¦¦   0x000100b7      660f1f8400..   nop word [rax + rax]
│   ¦ ¦¦¦   ; CODE XREF from fcn.0000fdf0 @ 0xff13(x)
│   ¦ ¦¦¦   0x000100c0      498b1424       mov rdx, qword [r12]
│   ¦ ¦¦¦   0x000100c4      8b35c2540300   mov esi, dword [0x0004558c] ; [0x4558c:4]=0
│   ¦ ¦¦¦   0x000100ca      31ff           xor edi, edi
│   ¦ ¦¦¦   0x000100cc      e81f1a0100     call fcn.00021af0
│   ¦ ¦¦¦   0x000100d1      ba05000000     mov edx, 5
│   ¦ ¦¦¦   0x000100d6      488d35ab5c..   lea rsi, str.error_waiting_for__s ; 0x35d88 ; "error waiting for %s"
│   ¦ ¦¦¦   0x000100dd      31ff           xor edi, edi
│   ¦ ¦¦¦   0x000100df      4989c4         mov r12, rax
```

The other nearby calls (`dcgettext,error,fcn.00021cb0`) do not write to `r12` at all. Rather, they handle error messages and directory changes. `fcn.0001c150` only influences whether we reach the **execvp** call.

Only `fcn.00021af0` returns a value that becomes `r12` and is later passed to **execvp**. That's why we choose `fcn.00021af0` as the next function to analyze.

Inside `fcn.00021af0` there is only one call instruction (call `fcn.00034980`), and the function immediately returns the value left in `rax`. Since `fcn.0000fdf0` copies `rax` into `r12` right after calling `21af0`, we can confirm that whatever `fcn.00034980` returns ultimately becomes the argument vector passed to **execvp.**

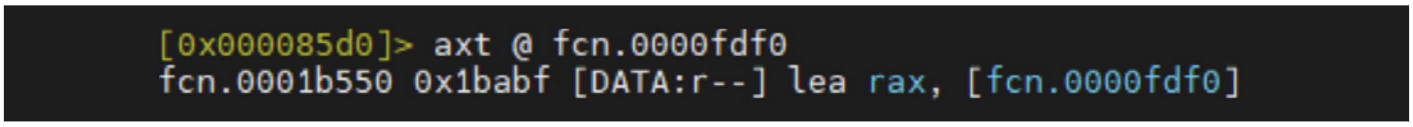


```
[0x000085d0]> pd @ fcn.00021af0
            ; XREFS(50)
┌ 142: fcn.00021af0 (int64_t arg2, int64_t arg3);
│ `- args(rsi, rdx) vars(8:sp[0x10..0x44])
│           0x00021af0      f30f1efa       endbr64
│           0x00021af4      4883ec48       sub rsp, 0x48
│           0x00021af8      89f0           mov eax, esi                ; arg2
│           0x00021afa      4889d6         mov rsi, rdx                ; arg3
│           0x00021afd      64488b1425..   mov rdx, qword fs:[0x28]
│           0x00021b06      4889542438     mov qword [var_38h], rdx
│           0x00021b0b      31d2           xor edx, edx
│           0x00021b0d      83f80a         cmp eax, 0xa
│       └─< 0x00021b10      0f842258feff   je 0x7338
│           0x00021b16      4889e2         mov rdx, rsp                ; int64_t arg3
│           0x00021b19      890424         mov dword [rsp], eax
│           0x00021b1c      c744240400..   mov dword [var_4h], 0
│           0x00021b24      48c7442408..   mov qword [var_8h], 0
│           0x00021b2d      48c7442410..   mov qword [var_10h], 0
│           0x00021b36      48c7442418..   mov qword [var_18h], 0
│           0x00021b3f      48c7442420..   mov qword [var_20h], 0
│           0x00021b48      48c7442428..   mov qword [var_28h], 0
│           0x00021b51      48c7442430..   mov qword [var_30h], 0
│           0x00021b5a      e8212e0100     call fcn.00034980
│           0x00021b5f      488b542438     mov rdx, qword [var_38h]
│           0x00021b64      64482b1425..   sub rdx, qword fs:[0x28]
│       ┌─< 0x00021b6d      7505           jne 0x21b74
│       │   0x00021b6f      4883c448       add rsp, 0x48
│       │   0x00021b73      c3             ret
```

Now I'm going to skip a few intermediate steps (because the process becomes recursive) just to validate that `r12` is indeed the pointer returned by `fcn.00034980`, and that it remains unchanged until it is later used as `argv` in the **execvp** call.

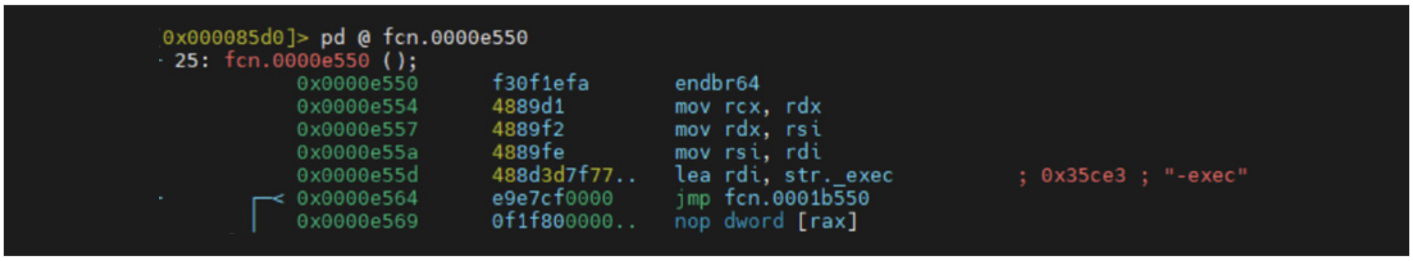
After validation, we want to see where the **execvp** caller function (`fcn.0000fdf0`) is coming from.

Using `axt @ fcn.0000fdf0` , we discover that this function is not called directly. Instead, its address is stored as a function pointer in another function (`fcn.0001b550`), which constructs an internal action node.



```
[0x000085d0]> axt @ fcn.0000fdf0
fcn.0001b550 0x1babf [DATA:r--] lea rax, [fcn.0000fdf0]
```

Tracing that backward reveals that `fcn.0001b550` is itself called from a small adapter function (`fcn.0000e550`) that hardcodes the string `"-exec"` into `rdi` and then jumps into the action constructor.



```
0x000085d0]> pd @ fcn.0000e550
· 25: fcn.0000e550 ();
           0x0000e550      f30f1efa       endbr64
           0x0000e554      4889d1         mov rcx, rdx
           0x0000e557      4889f2         mov rdx, rsi
           0x0000e55a      4889fe         mov rsi, rdi
           0x0000e55d      488d3d7f77..   lea rdi, str._exec          ; 0x35ce3 ; "-exec"
       ┌─< 0x0000e564      e9e7cf0000     jmp fcn.0001b550
       │   0x0000e569      0f1f800000..   nop dword [rax]
```

This conclusively ties the only **execvp** call in the binary to the parsing and evaluation of the `-exec` expression in the user's command line.

To sum it all up, we're left with a tedious process (as always with reverse engineering) until reaching that one function that matches our needs.

So after manually reversing a couple of already known LOLBins from GTFOBins to support my claim that this process reliably surfaced their execution paths I wanted to start coding.

At this point, my goal was very simple: to automatically find those known binaries (GTFOBins) with my tool, see whether it could rediscover known LOLBins, and potentially surface new ones that had not yet been documented.

Moving to Automation

At this point, the obvious question is: **where does AI come into play?** That's where things start to get interesting.

In this tool, AI is not used as a shortcut or a replacement for reverse engineering.
Instead, it acts as a reverse engineering assistant inside the process.

The core idea is simple: the tool does all the heavy lifting first. It collects data, builds context around each execution path, and only then sends a structured query to the AI. AI's role is limited to digesting that context and providing a decision: do the syscall parameters originate from command-line arguments, or not?

To support this, I implemented a helper class called **`AIUtils`**, which handles all communication with the AI backend: creating assistants, sending queries, and processing responses. In this project I used OpenAI, but the design is not tied to a specific provider. Any model with a Python API could be plugged in without changing the core analysis logic.

At the same time, I wanted a clean separation between reverse engineering logic and tooling. For that reason, I created another utility class called **`R2Utils`**. This class encapsulates all interactions with **`r2pipe`** and exposes the same primitives I used during the manual analysis phase, such as list functions, cross-reference discovery, and disassembly retrieval, along with additional helpers needed for automation.

With these building blocks in place, the focus shifted to the logic of the tool itself.

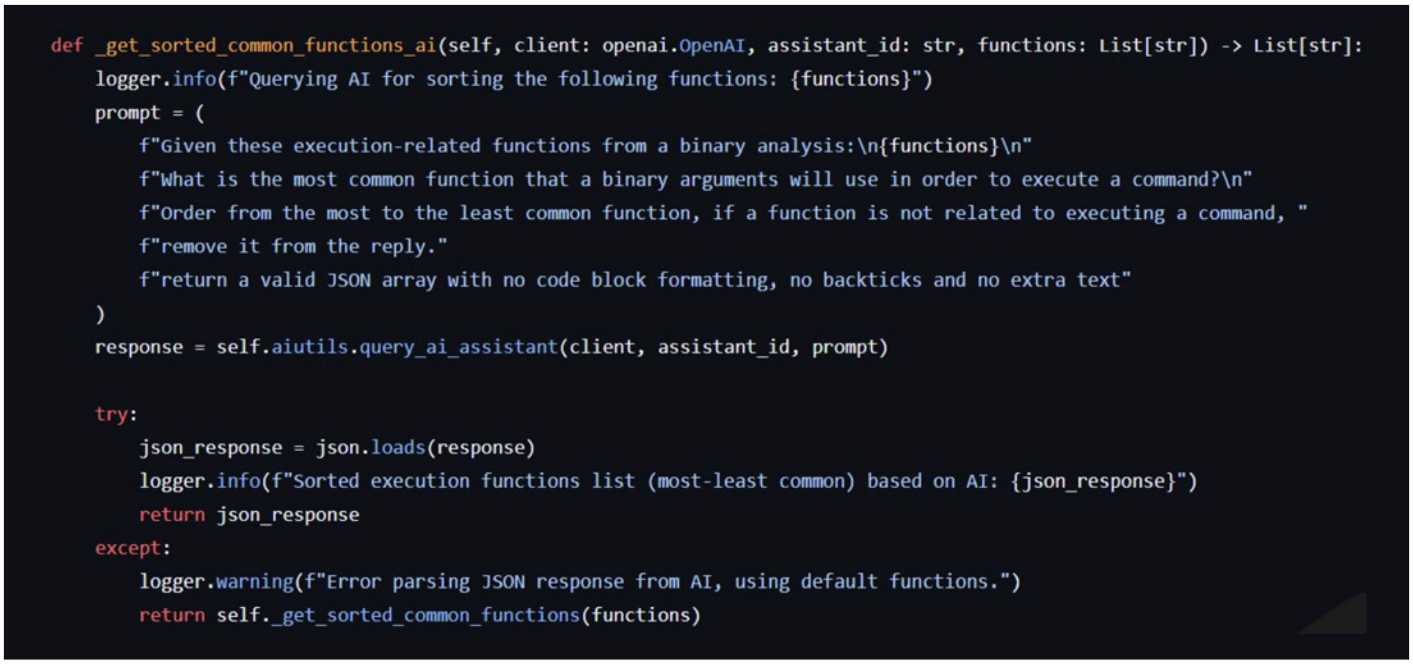
Automating the Manual Process

The plan was to build an automated tool that accepts either a single binary or an entire directory (such as `/usr/bin`) as input, and then follows the same reasoning process I previously applied by hand to each discovered executable.

First, each binary is opened using `r2pipe`, and full analysis is performed using radare2.

Next, the tool identifies candidate execution functions. Rather than relying solely on predefined assumptions, it can first leverage AI to analyze the full list of functions extracted from the binary and determine which ones are capable of executing external commands. AI is also used to sort and prioritize these candidates, allowing the analysis to focus on the most relevant execution paths early on.

This AI-driven classification is used as a discovery mechanism, especially for binaries that rely on less obvious helpers or undocumented execution wrappers.

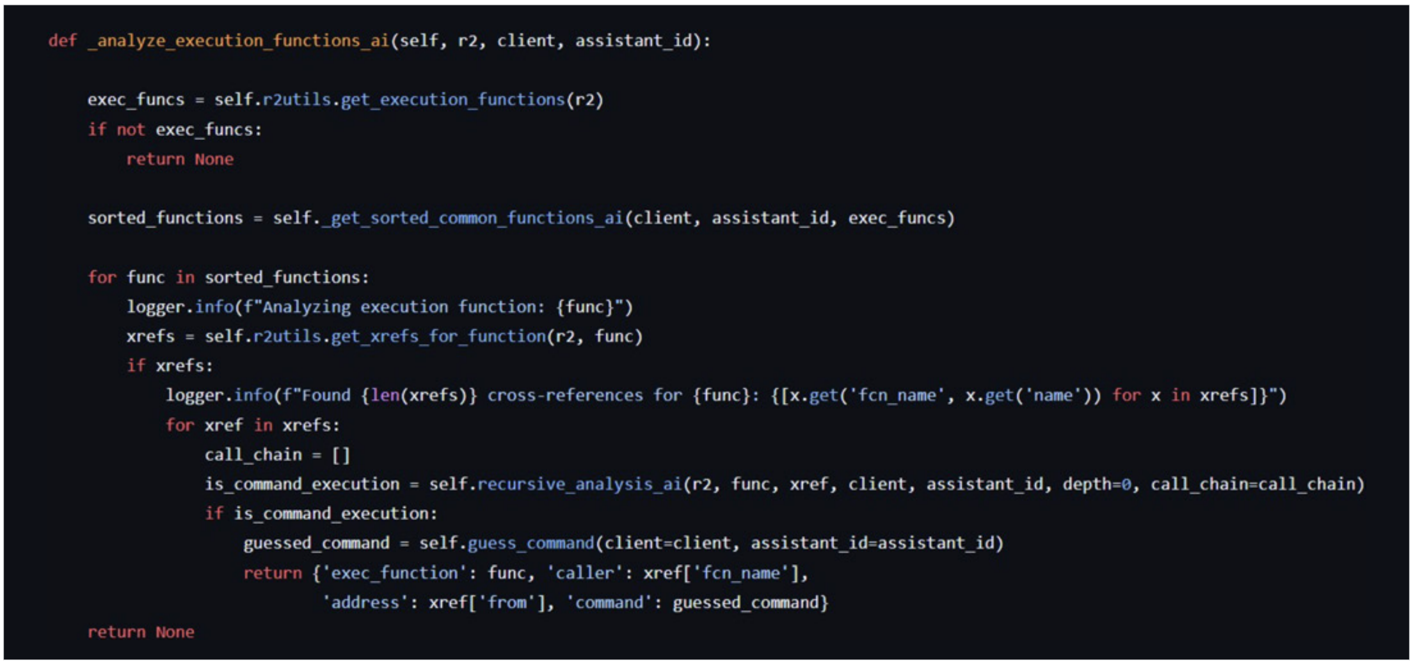


```python
def _get_sorted_common_functions_ai(self, client: openai.OpenAI, assistant_id: str, functions: List[str]) -> List[str]:
    logger.info(f"Querying AI for sorting the following functions: {functions}")
    prompt = (
        f"Given these execution-related functions from a binary analysis:\n{functions}\n"
        f"What is the most common function that a binary arguments will use in order to execute a command?\n"
        f"Order from the most to the least common function, if a function is not related to executing a command, "
        f"remove it from the reply."
        f"return a valid JSON array with no code block formatting, no backticks and no extra text"
    )
    response = self.aiutils.query_ai_assistant(client, assistant_id, prompt)

    try:
        json_response = json.loads(response)
        logger.info(f"Sorted execution functions list (most-least common) based on AI: {json_response}")
        return json_response
    except:
        logger.warning(f"Error parsing JSON response from AI, using default functions.")
        return self._get_sorted_common_functions(functions)
```

To keep the analysis grounded and fail-safe, the tool also maintains a predefined list of well-known execution primitives such as `execl, execvp, popen, fork`, and `system`. This list acts as a fallback and a safety net, ensuring that common and well-understood execution paths are always included, even if the AI classification is inconclusive or unavailable.

For each detected execution call, the tool builds a reverse call graph, starting from the execution point and tracing backward through the program until it reaches the entry point, typically `main` or an equivalent dispatcher function.



```python
def _analyze_execution_functions_ai(self, r2, client, assistant_id):

    exec_funcs = self.r2utils.get_execution_functions(r2)
    if not exec_funcs:
        return None

    sorted_functions = self._get_sorted_common_functions_ai(client, assistant_id, exec_funcs)

    for func in sorted_functions:
        logger.info(f"Analyzing execution function: {func}")
        xrefs = self.r2utils.get_xrefs_for_function(r2, func)
        if xrefs:
            logger.info(f"Found {len(xrefs)} cross-references for {func}: {[x.get('fcn_name', x.get('name')) for x in xrefs]}")
            for xref in xrefs:
                call_chain = []
                is_command_execution = self.recursive_analysis_ai(r2, func, xref, client, assistant_id, depth=0, call_chain=call_chain)
                if is_command_execution:
                    guessed_command = self.guess_command(client=client, assistant_id=assistant_id)
                    return {'exec_function': func, 'caller': xref['fcn_name'],
                            'address': xref['from'], 'command': guessed_command}
    return None
```

While building this call chain, the tool collects code snippets from each function along the path. These snippets preserve contextual disassembly around each call site and are later used for argument tracing and validation.

Once a complete call chain is constructed, the tool extracts the arguments passed to the execution function and traces their origin through the collected code. The goal is to determine whether those arguments ultimately originate from the program's input arguments.

If the argument can be traced back to user-controlled input, the tool flags the execution path as a potential command execution vector. If not, the result remains unproven.

The key point here is that the automated flow is not fundamentally different from the manual one. It is the same process, encoded and enforced programmatically.

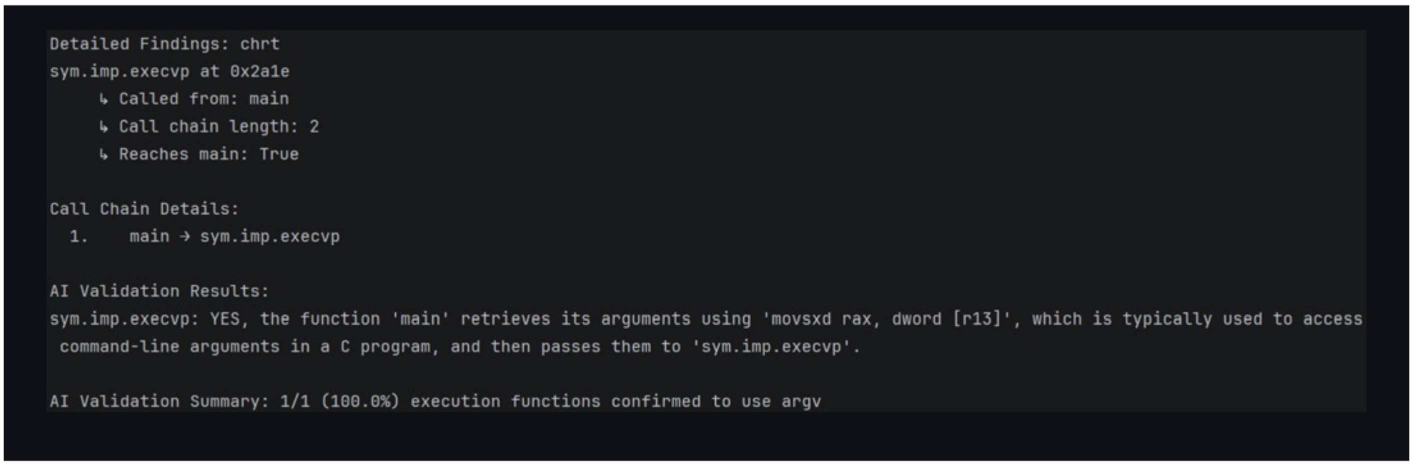
Scaling the Analysis

At some point, I needed to validate whether this approach actually worked at scale.

To do that, I set up a clean Ubuntu 22.04 environment and ran the tool against the entire `/usr/bin` directory, which contains roughly a thousand binaries. I launched the analysis overnight and waited for the results.

The moment of truth came the next morning. I filtered out binaries already documented in GTFOBins and focused on what remained. What surfaced were binaries that are not currently listed in the repository, yet exhibited execution paths consistent with LOLBin behavior that warranted further investigation.

Results of `/usr/bin/chrt`



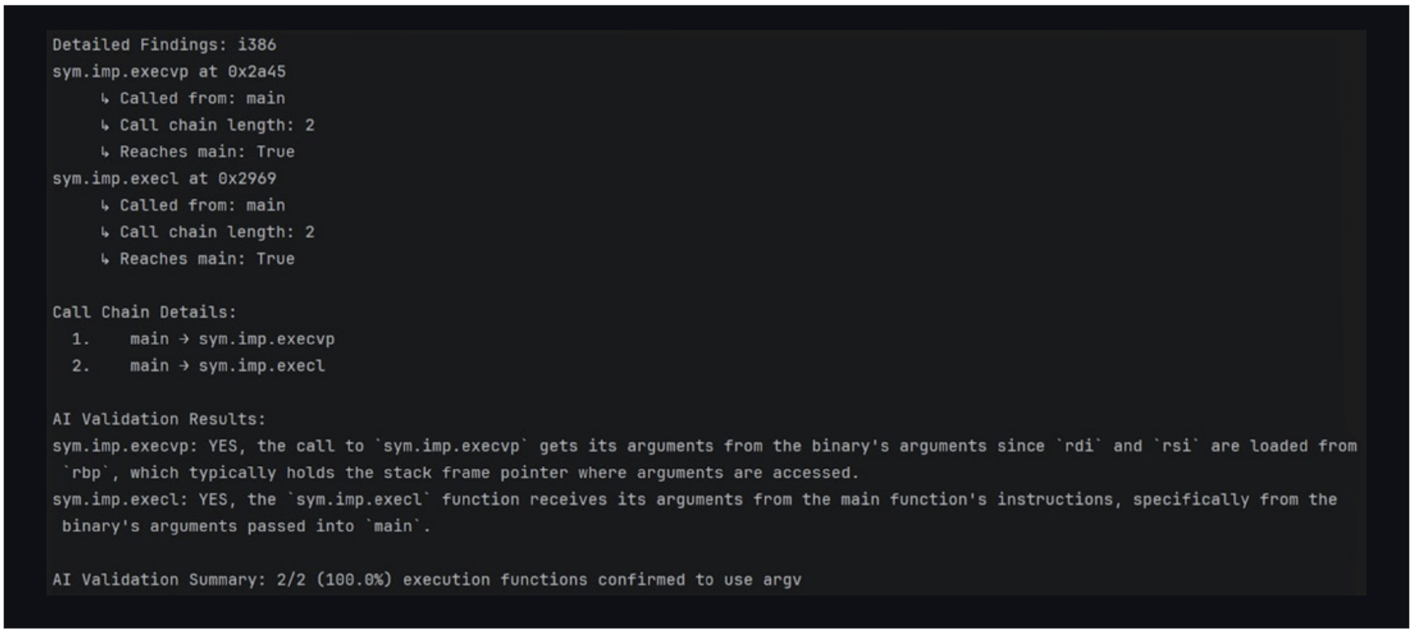
```
Detailed Findings: chrt
sym.imp.execvp at 0x2a1e
     ↳ Called from: main
     ↳ Call chain length: 2
     ↳ Reaches main: True

Call Chain Details:
  1.    main → sym.imp.execvp

AI Validation Results:
sym.imp.execvp: YES, the function 'main' retrieves its arguments using 'movsxd rax, dword [r13]', which is typically used to access
 command-line arguments in a C program, and then passes them to 'sym.imp.execvp'.

AI Validation Summary: 1/1 (100.0%) execution functions confirmed to use argv
```

Results of `/usr/bin/i386`



```
Detailed Findings: i386
sym.imp.execvp at 0x2a45
     ↳ Called from: main
     ↳ Call chain length: 2
     ↳ Reaches main: True
sym.imp.execl at 0x2969
     ↳ Called from: main
     ↳ Call chain length: 2
     ↳ Reaches main: True

Call Chain Details:
  1.    main → sym.imp.execvp
  2.    main → sym.imp.execl

AI Validation Results:
sym.imp.execvp: YES, the call to `sym.imp.execvp` gets its arguments from the binary's arguments since `rdi` and `rsi` are loaded from
 `rbp`, which typically holds the stack frame pointer where arguments are accessed.
sym.imp.execl: YES, the `sym.imp.execl` function receives its arguments from the main function's instructions, specifically from the
 binary's arguments passed into `main`.

AI Validation Summary: 2/2 (100.0%) execution functions confirmed to use argv
```

Getting this kind of validation was genuinely exciting. What started as a side research idea during day-to-day work turned into something tangible: a repeatable process that could surface interesting results without manual intervention.

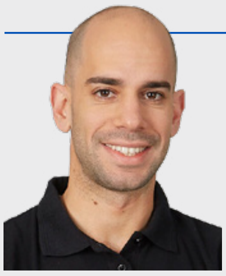
The Bigger Takeaway

While the findings themselves are interesting, they are not the most important outcome of this work.

What I found more meaningful is the process behind it: the ability to take existing knowledge and skills and amplify them through automation and AI. This is not just about discovering new privilege escalation paths or building another binary analysis tool. It's about learning how to extend your reach and efficiency by combining technical intuition with AI-driven reasoning.

If there is one takeaway I hope readers walk away with, it's that this mindset applies far beyond security research. The tools are already here. What matters is how creatively and responsibly we choose to use them.

About the author



Matan Abutbul is a Senior Security Researcher with over a decade of experience in cybersecurity research, penetration testing and incident response.
He's fascinated by how attackers think and operate at the intersection of threat research and defensive strategy, always exploring new ways to turn threat insights into stronger security.

Reach out to us with any questions about the research at labs@pentera.io.

About Pentera

PENTERA

Pentera is the market leader in AI-powered Security Validation, equipping enterprises with the platform to proactively test all their cybersecurity controls against the latest cyber attacks. Pentera identifies true risk across the entire attack surface, and automatically orchestrates remediation workflows to effectively reduce exposure. The company's security validation capabilities are essential for Continuous Threat Exposure Management (CTEM) operations. Thousands of security professionals around the world trust Pentera to close security gaps before threat actors can exploit them.

For more info, visit: pentera.io

PENTERA | Pentera Labs™ Research Series

LOLBins Against the Machine

www.Pentera.io