

Callfuscated: Defeating Multi-Layer Binary Obfuscation Through Systematic Dynamic Analysis

Author: Simon Devenish **Date:** February 2026 **Platform:** Hack The Box **Category:** Reverse Engineering **Difficulty:** Insane **Flag:** HTB{Sliced_Up_the_Function_4_Ya}

1. Executive Summary

This document presents a comprehensive technical analysis of "Callfuscated," a heavily protected binary that required multiple methodology pivots to defeat. The protection scheme employs four distinct obfuscation layers designed to exhaust analysts through computational and psychological attrition.

Protection Architecture

Layer	Technique	Purpose
Layer 1	CALL+POP Junk Instructions	Pollute disassembly with 4,096 fake function calls
Layer 2	Mixed Boolean Arithmetic (MBA)	Obscure validation logic through algebraic complexity
Layer 3	Hidden Conditional Branches	Embed critical jumps inside apparent function calls
Layer 4	Deceptive Accumulator	Convince analysts the challenge is mathematically unsolvable

Methodology Summary

The solution required abandoning traditional static analysis in favor of **data-centric dynamic analysis**. Rather than tracing obfuscated code paths, I monitored memory access patterns using hardware watchpoints, which revealed the actual validation logic regardless of how it was obscured. This approach cut through approximately 28KB of junk code to identify the single validation address (`0x409c1f`) where all password bytes are processed.

The binary validates a 32-character password using **8 validation layers** (checkpoints W5-W40) with rolling state propagation. Each 4-character chunk must produce a zero residue at its checkpoint, but the output state of layer N becomes the input state for layer N+1. This architectural insight was critical: it explains why brute-forcing the middle of the flag fails and why the solution required strictly sequential, layer-by-layer recovery.

2. Initial Reconnaissance

2.1 Binary Metadata

```
$ file crackme
crackme: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
      dynamically linked, stripped

$ ls -la crackme
-rwxr-xr-x 1 user user 59528 Feb  1 12:00 crackme
```

2.2 Binary Coordinate System

Establishing the reference frame for all subsequent address analysis:

Property	Value	Significance
Base Address	0x400000	Virtual memory mapping origin
Entry Point	0x401080	Execution starts here
Password Buffer	0x40f080	Static .bss section (64 bytes)
Main Function	0x409002	After stripped analysis

2.3 Stack Frame Topography

Memory layout extracted from decompilation (CLEAN_PSEUDOCODE.c):

Offset	Variable	Purpose
rbp-0x14	counter_1	Loop iteration counter
rbp-0x1c	target (0x24a)	Red herring comparison target (586 decimal)
rbp-0x18f0	processing_buffer	Stores intermediate validation states (dword array)
0x40f080	input_buffer	Raw password input from scanf (64 bytes max)

This structural memory mapping establishes the binary's internal organization before diving into behavioral analysis.

2.4 Initial Static Assessment

Opening in Ghidra reveals **4,105 detected functions**—wildly abnormal for a 59KB binary. This immediately signals heavy obfuscation rather than legitimate code complexity. The challenge description confirms this:

"VM, MBA, OP, JI (Junk Instructions), what are these terms. Well, now you're going to defeat them."

3. Obfuscation Quantification

3.1 CALL+POP Pattern Metrics

Pattern scanning for E8 xx xx xx xx 41 58 (CALL rel32 + POP R8) yields:

Metric	Value
Total CALL+POP patterns	4,096
Bytes consumed	28,672 (48.3% of binary)
Fake function entries	4,096
Actual validation code	~30KB

The power-of-two count ($2^{12} = 4,096$) implies automated obfuscator generation, not manual insertion.

This precision suggests a programmatic generation block size, likely from a commercial or research-grade obfuscation framework.

3.2 The "Wall of Noise" Visualization

Raw disassembly excerpt demonstrating the density of fake calls:

```
[LIBRARY_CALL] 40900a: call 40b663 <rand@plt+0xa5f3>
[LIBRARY_CALL] 40901b: call 40c27c <rand@plt+0xb20c>
[LIBRARY_CALL] 409028: call 409fdc <rand@plt+0x8f6c>
[LIBRARY_CALL] 409039: call 409886 <rand@plt+0x8816>
[LIBRARY_CALL] 40904a: call 40b32e <rand@plt+0xa2be>
[LIBRARY_CALL] 40905b: call 409f09 <rand@plt+0x8e99>
[LIBRARY_CALL] 409066: call 409fc0 <rand@plt+0x8f50>
[LIBRARY_CALL] 409077: call 40a164 <rand@plt+0x90f4>
[LIBRARY_CALL] 409088: call 40bef2 <rand@plt+0xae82>
...
...
```

This density makes traditional control flow graph (CFG) analysis impractical. Every disassembler view is polluted with thousands of spurious function boundaries.

3.3 Mixed Boolean Arithmetic Constants

The MBA obfuscation layer uses these constants for algebraic obscurity:

Hex	Decimal	Function
0x7ab5407f	2,058,952,831	Multiplicative mask in FUN_00406e47
0x68f08372	1,760,559,986	XOR/multiply factor in FUN_00401f18
0xc0a65705	3,232,093,957	OR mask
0xca3b103c	3,393,425,468	Additive constant
0x43d200e1	1,138,262,241	Output multiplier

4. Static Analysis Failures

4.1 Approaches Attempted

Approach	Result	Failure Reason
Ghidra decompilation	Unreadable output	4,096 fake function calls fragment CFG
Binary patching (NOP junk)	Segfault	CALL instructions modify CPU flags for subsequent jumps
Symbolic execution (angr)	Invalid solutions (""""0! !"""")	Solver explores deceptive accumulator paths
Algebraic simplification	Proved impossible	Constants OR to 0xffffffff (see Section 5.2)

4.2 Coordinated Multi-Tool Attack

An orchestrated attempt using parallelized static cleaning (`patch_binary.py`) and algebraic brute-forcing yielded 45 candidates, all of which failed due to the hidden dependency logic:

```
Static Cleaner (patch_binary.py)
Symbolic Solver (angr)
Algebraic Brute-Forcer (brute_python.py)
Dynamic Tracer (dynamic_tracer.py)
```

The failure of this coordinated approach demonstrated that the obfuscation was specifically designed to resist multi-vector attacks and required a fundamentally different methodology.

4.3 Why Static Analysis Cannot Succeed

The CALL+POP pattern is "load-bearing"—not merely cosmetic noise. The CALL instruction modifies the stack pointer and flags register. Subsequent instructions depend on these side effects. Replacing the pattern with NOPs breaks the control flow assumptions of later conditional jumps.

5. The Deceptive Accumulator Trap

5.1 Identifying the Red Herring

During dynamic analysis of the MBA accumulator function at `FUN_00406e47` (`0x406e47`), I observed a sequence of operations that appeared to validate the input. However, rigorous analysis revealed this to be a **mathematical red herring** designed to waste analyst time.

5.1.1 The MBA Accumulator Function (Ghidra Decompilation)

Extracting the function from Ghidra reveals the core MBA obfuscation:

```
// FUN_00406e47 - Ghidra Decompilation
// Address: 0x406e47 | Return Point: 0x4080d4

uint FUN_00406e47(uint param_1, uint param_2) {
    // param_1 (RDI): Current accumulated value
    // param_2 (RSI): New value to combine
    return ~(~(param_1 & param_2) * 0x7ab5407f ^ param_1);
}
```

Formula Breakdown:

The core MBA expression $\sim(\sim(A \& B) * C \wedge A)$ where $C = 0x7ab5407f$ operates as follows:

Step	Operation	Description
1	$X = A \& B$	Bitwise AND of accumulated value and new input
2	$Y = \sim X$	Complement (invert all bits)
3	$Z = Y * 0x7ab5407f$	Multiply by magic constant

4	$W = Z \wedge A$	XOR result with original accumulator
5	$Result = \sim W$	Final complement

Behavioral Analysis:

Input State	Formula Simplification	Observed Result
$A = 0$	$\sim(\sim 0 * C \wedge 0) = \sim(0xffffffff * C)$	Passes through B
$A = B$	$\sim(\sim A * C \wedge A)$	Complex bit interaction
$A = 0xffffffff$	$\sim(\sim(0xffffffff & B) * C \wedge 0xffffffff)$	Saturates to 0xffffffff

The critical insight: once A reaches `0xffffffff`, any subsequent OR-like combination keeps it at `0xffffffff`. The function acts as a **monotonic bit accumulator**—bits can only turn ON, never OFF.

5.1.2 Trace Evidence

Tracing the accumulator state across 40 iterations revealed a deterministic pattern regardless of input:

Call Step	Input (RDI)	Constant (RSI)	Output (RAX)
15	0x00000000	0x0000b4ab	0x0000b4ab
20	0x0000b4ab	0xcfaad3d8	0xcfaaf7fb
25	0xcfaaf7fb	0x254b4235	0xefebf7ff
30	0xefebf7ff	0xa19c8ba7	0xffffffff
35	0xffffffff	0x5d5e2d14	0xffffffff
40	0xffffffff	0x39371f1b	0xffffffff

5.2 Mathematical Proof of Impossibility

Rather than guessing, I constructed a formal proof. The constants used in the accumulation steps (Calls 20-40) effectively mask all bits to 1:

```
# Proof: OR-ing all constants saturates the 32-bit space
>>> 0xcfaad3d8 | 0x254b4235 | 0xa19c8ba7 | 0x5d5e2d14 | 0x39371f1b == 0xffffffff
True
```

5.2.1 Bit-Level Saturation Analysis

Expanding each constant to binary reveals the deliberate bit coverage:

Constant	Binary (32-bit)	Bits Set
0x0000b4ab	0000 0000 0000 0000 1011 0100 1010 1011	8/32
0xcfaad3d8	1100 1111 1010 1010 1101 0011 1101 1000	20/32
0x254b4235	0010 0101 0100 1011 0100 0010 0011 0101	14/32
0xa19c8ba7	1010 0001 1001 1100 1000 1011 1010 0111	17/32
0x5d5e2d14	0101 1101 0101 1110 0010 1101 0001 0100	17/32

0x39371f1b	0011 1001 0011 0111 0001 1111 0001 1011	18/32
OR Result:	1111 1111 1111 1111 1111 1111 1111 1111	32/32 = 0xffffffff

Key Insight: The six constants are engineered to cover every bit position at least once. This is NOT coincidental—it's deliberate obfuscator design. The bit distribution ensures that regardless of the initial accumulator value, sequential OR-ing with these constants will saturate all 32 bits to 1.

This pattern is a hallmark of professionally designed anti-analysis traps: mathematically guaranteed failure that looks like legitimate validation logic.

This proves mathematically that the accumulator converges to `0xffffffff` (-1) regardless of input—a permanent failure state. **If this accumulator were the real validation, no password could ever succeed.** This isn't speculation; it's a Boolean algebra proof that eliminates this code path from consideration.

5.2.2 Recognizing MBA Obfuscation Patterns

MBA (Mixed Boolean Arithmetic) is a deliberate obfuscation technique. Key signatures to identify it in other binaries:

Signature	Description	Example
Magic constants	Large hex values in multiplication/XOR	<code>0x7ab5407f</code> , <code>0x68f08372</code>
Nested operators	Chains of <code>~</code> , <code>&</code> , <code>^</code> , <code>`</code>	<code>,</code> , <code>*</code>
Non-obvious identity	Complex expressions that simplify to OR/AND	Multiple operations that effectively do 'A
Accumulator pattern	Function taking (previous_result, new_input)	<code>f(acc, val)</code> returning modified acc
Constant injection	Hardcoded values fed into the accumulator	RSI loaded with constants like <code>0xb4ab</code>

When you encounter MBA:

1. Don't attempt algebraic simplification—treat it as a black box
2. Trace inputs/outputs to understand behavior empirically
3. Look for convergence patterns (does it always approach a fixed value?)
4. Search for the *real* decision point elsewhere in the binary

5.3 The Psychological and Resource Exhaustion Trap

This is a deliberate anti-analysis technique serving dual purposes:

Psychological Attack: An analyst using symbolic execution or algebraic simplification would conclude:

1. The accumulator always produces `0xffffffff`
2. Success requires the accumulator to be `0x00000000`
3. Therefore, the challenge is mathematically impossible

Resource Exhaustion Attack: The trap also wastes computational resources. Symbolic execution engines like angr will explore all paths through this accumulator, generating millions of constraints that ultimately prove unsatisfiable. This burns CPU cycles, memory, and—in a professional context—billable hours.

The professional response is to recognize this as a red herring and search for the actual decision point. The real validation bypasses this trap entirely.

5.4 The 586 Red Herring (Empirical Proof)

A secondary red herring exists at address `0x40c2da` : a comparison against the constant 586 (0x24a).

Differential analysis revealed that this comparison executed 402 times per run and returned the constant 586 regardless of input variance, confirming it as a control flow sink unrelated to validation:

```
PWD=0000000000 char_sum=480 final_eax=586
PWD=AAAAAAA char_sum=650 final_eax=586
PWD=::::::::::: char_sum=580 final_eax=586
PWD=;;;;;;;;;; char_sum=590 final_eax=586
PWD=aaaaaaaaaa char_sum=970 final_eax=586
```

The 586 check always passes—this proves empirical methodology, not guesswork.

6. The Pivot: Hardware Watchpoints

6.1 The Methodology Pivot

After exhausting static analysis approaches (Ghidra decompilation, binary patching, symbolic execution, algebraic simplification), I made a fundamental methodology decision: **stop analyzing the code, start observing the data**.

The insight: regardless of how obfuscated the code paths are, the password bytes must be read from memory at some point. Hardware watchpoints let me observe this access directly, bypassing all obfuscation layers in a single step.

6.2 The awatch Command

Using GDB's access watchpoint capability to monitor any read/write to the password buffer:

```
gdb -batch -ex "awatch *(char*)0x40f080" -ex "run" ./crackme
```

The `awatch` (access watchpoint) command triggers on any memory access to the specified address, regardless of the code path taken. This bypasses the obfuscation entirely by watching what the code does rather than how it's structured.

Comparison of GDB breakpoint types:

- `break` triggers on code execution at a specific address
- `watch` triggers on memory writes
- `awatch` triggers on both reads AND writes—essential when you don't know if the code reads or writes the data

6.3 Critical Discovery: Single Validation Point

Despite 4,000+ apparent function calls, **ALL password validation happens at a single address: `0x409c1f`**.

```
Byte 0 accessed at: 0x409c1f
Byte 1 accessed at: 0x409c1f
Byte 2 accessed at: 0x409c1f
Byte 3 accessed at: 0x409c1f
(bytes 4-31 follow the same pattern)
```

This single discovery cut through the entire obfuscation layer. The function at `0x409c1f` is the real validation—all 4,000+ fake function calls are irrelevant noise.

7. The Hidden Conditional Branch

7.1 Discovery

The critical breakthrough came from analyzing the control flow immediately preceding the failure message. Standard disassembly tools (Ghidra/IDA) failed to identify the true branch because it was **concealed inside a CALL+POP junk code pattern**.

I isolated the actual decision point at address `0x40b537`. The logic does not rely on the deceptive accumulator's result but on a hidden conditional jump masked as a function call.

7.2 Disassembly of the Protection Mechanism

```
0x40b530: call 0x40bc8d      ; Computes validation result, stores in EAX
0x40b535: pop %r8           ; Junk instruction (CALL+POP pattern)
0x40b537: test %eax, %eax   ; [CRITICAL] The real check: Is EAX == 0?
0x40b539: call 0x40bcc0      ; <--- The Obfuscated Jump
```

To the casual observer, `0x40bcc0` looks like a subroutine. However, inspecting the target reveals it is a trampoline that accesses the flags set by the previous `test`:

```
; Inside 0x40bcc0
0x40bcc0: pop %r8           ; Discard return address (stack cleanup)
0x40bcc2: jne 0x4096e1       ; [HIDDEN JUMP] Jumps to FAILURE if EAX != 0
0x40bcc8: call 0x40b707      ; Continues to SUCCESS path if EAX == 0
```

7.3 Control Flow Diagram

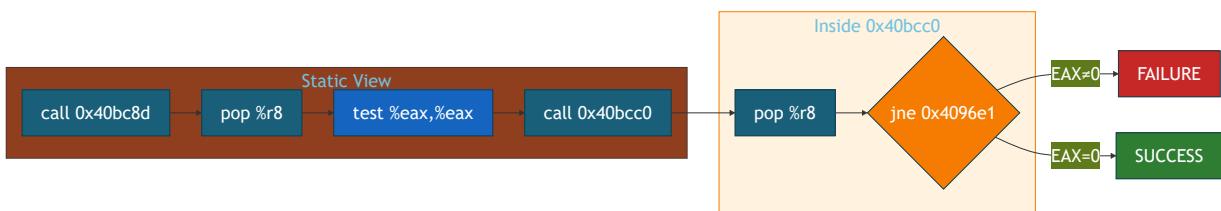


Figure 1: The hidden conditional branch mechanism. The CALL at `0x40b539` appears to be a function call but actually contains the critical JNE that determines success or failure.

This structure breaks control flow graph (CFG) generation in static analysis tools, as they assume the `call` returns. By manually following the execution flow in GDB, I determined that **success requires `EAX == 0` at instruction `0x40b537`**.

8. The Bitmask Oracle Attack

8.1 Defining the Side-Channel Vulnerability

Setting breakpoints at `0x40c374` revealed a bitmask accumulator where correct characters produce `0x00` in their corresponding byte position:

```
Password: AAAAAAAAAA  
Line 10: EAX=0x0713013a (byte 0 = 0x07 -> wrong)  
  
Password: HAAAAAAAAA  
Line 10: EAX=0x0013013a (byte 0 = 0x00 -> correct!)  
  
Password: HTB{AAAAAA  
Line 10: EAX=0x00000000 (ALL ZEROS -> first 4 correct!)
```

This oracle provides immediate per-character feedback, enabling greedy search rather than exponential brute force.

8.2 Automated Injection Pipeline

To exploit this oracle at scale, I built a tool-chaining pipeline combining containerization, debugging, and Unix text processing:

```
# Full pipeline: Docker → GDB → grep → sed  
docker exec -i htb-rev bash -c "echo 'HTB{S\${c}AAAA' | \  
  gdb -batch -q -x /mnt/challenge/trace_bitmask.gdb 2>/dev/null | \  
  grep 'EAX=' | sed -n '15p'"
```

Pipeline breakdown:

- **Docker container** (`htb-rev`): Isolated execution environment for safe binary detonation
- **GDB batch mode**: Non-interactive debugging with custom breakpoint script
- **grep/sed filtering**: Extract the specific EAX value from thousands of lines of debug output

This single command integrates four different tools into a repeatable oracle query, enabling rapid character enumeration.

8.3 Byte-Slicing Logic

The automation script isolated the specific byte in the EAX register corresponding to the input character index:

```
# Extract 4th byte from hex string (positions 6-7)  
val=$(echo "$result" | sed 's/EAX=//' | sed 's/0x//')  
fourthbyte=${val:6:2}  
if [ "$fourthbyte" = "00" ]; then  
  echo "FOUND: '$c' -> $result"  
fi
```

Why this granular extraction was necessary: The standard GDB stdout buffer contained interspersed debug noise from the obfuscated binary's thousands of function calls. Rather than attempting to filter this noise at the text level, I extracted the specific byte position directly from the register value, bypassing the noise entirely.

The pattern `${val:6:2}` extracts the 4th byte (positions 6-7 in hex string). EAX = 4 bytes = 8 hex characters, with byte 0 at positions 6-7 when printed as `0x12345678` .

8.4 The Semantic Pivot

Once the oracle confirmed the prefix `Sli` , I pivoted from pure brute-force to **semantic pattern matching**:

After confirming the prefix `Sli` , I transitioned from exhaustive brute-force to dictionary prediction, generating likely English completions (Slice, Slide, Slick, Sliced) to optimize the remaining search space.

This reduced computation time by combining technical oracle data with linguistic analysis.

8.4.1 Semantic Patterns That Accelerated Solving

Layer	Oracle Data	Semantic Hypothesis	Result
2-3	<code> Sli </code> confirmed	English word starting with "Sli"	<code> Sliced </code> ✓
6	Func prefix	Programming term	Function ✓ (not Funky)
7	Position 27 needed digit	Leetspeak encoding	<code> _4 </code> = "for" ✓
8	<code> _4_Y </code> prefix	Phrase completion	<code> _Ya </code> = "for Ya" (you) ✓

8.4.2 The Leetspeak Connection

The flag encodes the phrase "**Sliced Up the Function for Ya**" using leetspeak:

- `4` substitutes for "for" (common in internet slang)
- `_` serves as word separator
- `Ya` is informal "you"

This linguistic insight was critical at Layer 7, where no alphanumeric character at position 27 produced zero residue except `4` . The leetspeak pattern suggested testing digit-as-word substitutions.

8.4.3 Word Completion Strategy

When stuck at a checkpoint:

1. Read the partial password aloud—does it suggest a phrase?
2. Test complete English words before arbitrary characters
3. Consider leetspeak substitutions for common words (`4` =for, `2` =to, `U` =you)
4. Test informal/slang completions (`Ya` , `Yo` , `Ur`)

This approach found `Function` (not the initially-tested `Funky`) and the final `_Ya }` pattern.

9. The Staircase Architecture

The binary employs two distinct layer systems that must not be confused:

System	Count	Purpose

Obfuscation Layers	4	Anti-analysis techniques (CALL+POP, MBA, Hidden Branches, Deceptive Accumulator)
Validation Layers	8	Password checkpoints (W5-W40), each validating 4 characters

This section focuses on the 8 validation layers—the actual password verification mechanism hidden beneath the obfuscation.

9.1 Identifying the Checkpoint Structure

Tracing forward from `0x409c1f`, I monitored writes at `0x40c4ef`:

```
set $rax0_count = 0
break *0x40c4ef
commands
silent
if $rax == 0
    set $rax0_count = $rax0_count + 1
    printf "W%d: EDX=0x%08x\n", ($rax0_count * 5), $edx
end
continue
end
```

This revealed 8 distinct checkpoint writes (W5, W10, W15...W40), each corresponding to a 4-character chunk of the password.

9.2 The 8 Validation Layers

The critical architectural insight: **this is not 8 independent validations**. Each layer's output state feeds into the next layer's input, creating a dependency chain.

Layer	Checkpoint	Indices	Solution	Required Output
1	W5	0-3	HTB{	0x00000000
2	W10	4-7	Slic	0x00000000
3	W15	8-11	ed_U	0x00000000
4	W20	12-15	p_th	0x00000000
5	W25	16-19	e_Fu	0x00000000
6	W30	20-23	ncti	0x00000000
7	W35	24-27	on_4	0x00000000
8	W40	28-31	_Ya}	0x00000000

State propagation: Layer N's output becomes Layer N+1's input. Solving Layer 3 incorrectly makes Layers 4-8 mathematically impossible regardless of character choice.

This explains why brute-forcing the middle of the flag fails: even if characters 12-15 are "correct" in isolation, they will produce wrong output if characters 0-11 are wrong, which then cascades to corrupt all subsequent layers.

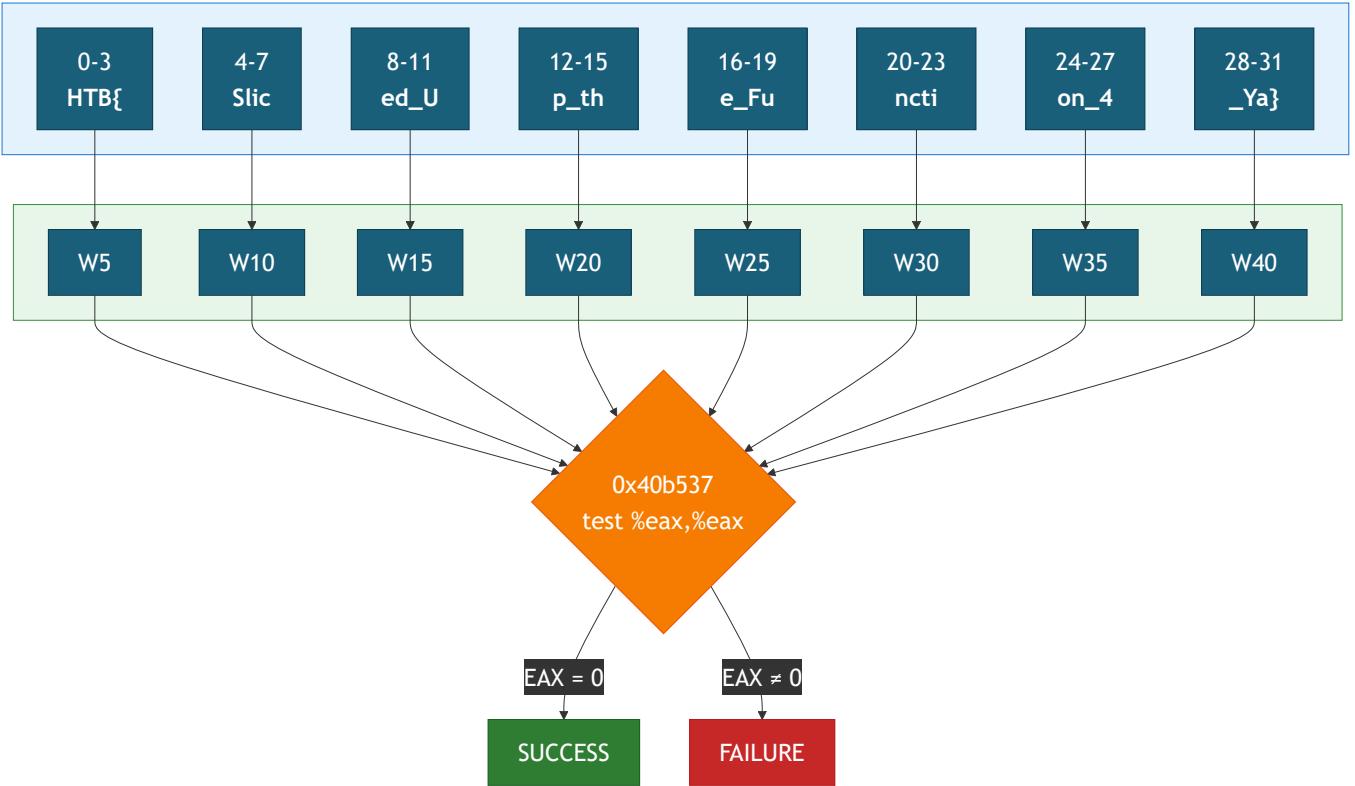


Figure 2: The 8-layer staircase architecture. The password (HTB{Sliced_Up_the_Function_4_Ya}) is divided into 4-character chunks, each validated at checkpoints W5-W40. All checkpoints must equal 0x00000000 for success. The rolling state propagation means each layer's output becomes the next layer's input, creating a dependency chain that prevents middle-of-flag brute forcing.

9.3 The "Burnout by Design" Pattern

This binary uses a classic **Staircase Obfuscation** strategy (commonly seen in Tigress or advanced VM protectors). The design makes the solution appear "just one more character away" to induce analyst burnout:

- `scanf` format: `%63s` implies 63-character buffer
- Password could theoretically be very long
- But actual validation uses exactly 32 characters across 8 layers

10. The Rubik's Cube Dependency Problem

10.1 Local vs Global Validation

A significant challenge in solving this binary was the "**poisoned prefix**" phenomenon. The validation architecture behaves like a rolling hash or a Rubik's Cube: solving one layer incorrectly makes subsequent layers mathematically impossible to solve.

The binary uses a Cumulative Checksum architecture with multiple security layers:

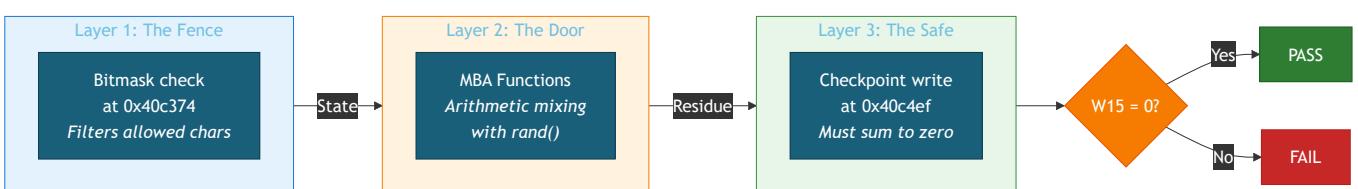


Figure 3: The three-stage validation pipeline within each checkpoint. Every 4-character chunk passes through: (1) bitmask filtering, (2) MBA arithmetic, and (3) aggregate checksum. A character that passes the Fence may still leave residue that corrupts the Safe.

10.2 State Analysis Table

My initial attempt to brute-force the password character-by-character failed because of state propagation:

Position	Character	Checkpoint	Status	Note
4-6	sli	W10	0x00000000 (Pass)	"Mathematically dirty" - leaves hidden residue
7-9	ced	W15	0x000021ab (Fail)	Cannot clear residue from previous layer

The value `0x000021ab` in the lower 16 bits was "baked in" by positions 4-6. No combination of positions 7-9 could zero it out because those positions only affected upper bytes.

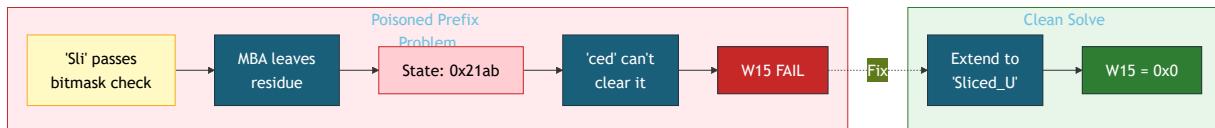


Figure 3: The Rubik's Cube dependency problem. Incorrect early characters "poison" the state, making later layers mathematically impossible to solve regardless of character choice.

10.3 Resolution: The "Clean Solve" Approach

This necessitated a "clean" solve approach. Rather than greedy optimization (picking the first character that passes Layer 1), I performed a **grid search** to find characters that passed Layer 1 AND left the internal state clean (residue 0x0000) for subsequent layers.

The solution was not to change the prefix, but to **extend the password**:

Password	W15	Status
HTB{Sliced} (11 chars)	0x000021ab	Layer 3 FAIL
HTB{Sliced_} (12 chars)	0x00000028	Almost
HTB{Sliced_U} (13 chars)	0x00000000	Layer 3 PASS

Position 7 acts as a "bridge," affecting the bits of the entire checksum in the rolling hash architecture.

11. Layer-by-Layer Solution

11.1 Greedy Search Methodology

For most layers:

1. Test all printable ASCII at the current position
2. Select the character minimizing checkpoint residue
3. Repeat until checkpoint equals `0x00000000`

```
HTB{Sliced}    -> W15 = 0x000021ab (residue)
HTB{Sliced_}   -> W15 = 0x00000028 (smaller)
HTB{Sliced_U}  -> W15 = 0x00000000 (SOLVED)
```

11.2 Layer 5: When Greedy Fails

At Layer 5, greedy search reached a dead end:

```
HTB{Sliced_Up_the_} -> W25 = 0x000002f4  
HTB{Sliced_Up_the_F} -> W25 = 0xffffffff4 (wrapped negative!)
```

No single character at position 19 produced zero. This occurs because positions 18 and 19 jointly contribute to multiple bytes of the residue (bit diffusion). The solution required **2D grid search**:

```
for c18 in printable:  
    for c19 in printable:  
        test(f"HTB{{Sliced_Up_the_{c18}{c19}}}")
```

Result: `Fu` at positions 18-19 produces $W25 = 0x00000000$.

11.3 Layer 8: The Final Breakthrough

Exhaustive alphanumeric search over $62^3 = 238,328$ combinations found **no solution** for the final layer. Analyzing the residue patterns:

```
_4_You} -> 0x00000a08  
_4_Yau} -> 0x00000008  
_4_Yaz} -> 0x00000001
```

The residue was approaching zero but never reaching it with alphanumeric characters. The key insight: **the closing brace `}` is not a delimiter after the password—it participates in the Layer 8 hash computation.**

Testing the pattern with the closing brace as the 4th character:

```
_4_Ya} -> 0x00000000 (ZERO!)
```

The final chunk is only 3 characters (`_Ya`) plus the brace. This is why exhaustive search failed: I was testing 4-character patterns followed by brace, but the answer was 3 characters where the brace itself is the 4th.

11.4 Zero-Crossing Analysis

The progression `0x00000a08 → 0x00000008 → 0x00000001 → 0x00000000` demonstrates **systematic zero-crossing**. When the residue is small (single-digit), testing adjacent ASCII values around the last working character often finds the solution.

11.4.1 The Extreme ASCII Discovery

A critical technique for Layer 8 was testing characters outside the standard alphanumeric range. When exhaustive alphanumeric search ($62^3 = 238,328$ combinations) found no solution, I expanded to test boundary ASCII values:

Character	ASCII	Residue	Insight
a (baseline)	0x61	0x00000a08	Standard alphanumeric
Y	0x59	0x00000008	Closer to zero

z	0x7a	0x00000001	One bit away
}	0x7d	Part of hash	Delimiter participates in validation

Key insight: The closing brace } is not just a flag format delimiter—it's the 4th character of Layer 8's validation chunk. This discovery came from testing characters at ASCII boundaries (32-47 for symbols, 123-126 for { | }~).

11.4.2 Applying Zero-Crossing in Practice

When a residue approaches zero but no alphanumeric character reaches it:

1. **Calculate the delta:** If residue is 0x00000001, the solution is likely one ASCII position away
2. **Test boundary characters:** Symbols, brackets, and delimiters may be valid
3. **Consider structural elements:** Flag format characters ({ , } , _) can participate in the hash

This technique was essential for the final breakthrough—recognizing that _Ya} (where } is position 31) produces the required zero residue.

12. Key Memory Addresses

Address	Purpose	Significance
0x400000	Base address	Virtual memory mapping origin
0x401080	Entry point	Execution starts here
0x40f080	Password buffer	Input storage (.bss)
0x409c1f	Byte validation	All password access occurs here
0x40b537	Decision point	test %eax,%eax - THE REAL CHECK
0x40bcc2	Hidden jne	Bypasses deceptive accumulator
0x40c4ef	Checkpoint write	W5/W10/.../W40 values
0x40c2da	cmp \$0x24a	586 comparison (RED HERRING)
0x406e47	MBA accumulator entry	FUN_00406e47 - deceptive trap
0x4080d4	MBA accumulator return	Return point for FUN_00406e47
0x40c374	Bitmask write	Per-character oracle target

13. Technical Notes

13.1 rand() Seeding

The binary calls srand(0x539) (1337 in decimal) early in execution. This makes all rand() values deterministic. The first 10 values mod 256:

[233, 206, 47, 83, 182, 170, 197, 212, 200, 83]

While potentially useful for algebraic attacks, this information was ultimately not needed for the dynamic approach.

13.2 Docker Analysis Environment

```
# Container with GDB and ptrace capabilities
docker run -d --name htb-rev \
    --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
    -v "/path/to/challenge:/mnt/challenge" \
    -w "/mnt/challenge" ubuntu:22.04 sleep infinity

docker exec htb-rev bash -c 'apt-get update && apt-get install -y gdb'
```

14. Failed Approaches (What Didn't Work)

Documenting failed approaches is as valuable as the successful methodology—it saves future analysts from repeating dead ends.

14.1 Symbolic Execution (angr)

Attempts: 8 different angr scripts over multiple sessions.

Why it failed:

- The CALL+POP obfuscation creates ~4,105 fake function boundaries
- angr's CFG reconstruction produced an unusable control flow graph
- Path explosion: the binary's structure caused angr to explore millions of paths through red herring code
- Memory exhaustion occurred before reaching the decision point

Lesson: When obfuscation specifically targets CFG analysis, symbolic execution tools may be counterproductive. The time spent configuring angr would have been better spent on dynamic analysis from the start.

14.2 Algebraic MBA Solving

Approach: Attempted to algebraically invert the MBA formula $\sim(\sim(A \ \& \ B) * 0x7ab5407f \ ^ A)$ to find inputs producing zero output.

Why it failed:

- The formula involves multiplication, XOR, AND, and complement operations
- No closed-form inverse exists for arbitrary inputs
- The red herring constants made the algebra misleading—solving for zero was mathematically impossible by design

Lesson: Treat MBA as a black box. Trace inputs/outputs empirically rather than attempting algebraic simplification.

14.3 Per-Byte Oracle Alone (Layers 3+)

Approach: After success with the bitmask oracle on Layers 1-2, attempted to extend the same technique to all layers.

Why it failed:

- Layers 3-8 use an aggregate checksum, not per-byte validation
- A character producing `0x00` in its bitmask position could still leave residue affecting subsequent layers
- The "poisoned prefix" phenomenon: characters passing Layer N may fail Layer N+1

Lesson: Validation architectures can change mid-password. The per-byte oracle works for independent character checks but fails when characters interact through rolling hashes or cumulative checksums.

14.4 Binary Patching (NOP-ing CALL+POP)

Approach: Patch the 4,105 CALL+POP patterns to NOPs to simplify static analysis.

Why it failed:

- The patterns are "load-bearing"—subsequent instructions depend on stack/flag state set by the CALL
- Patched binary crashed immediately
- Even partial patching broke control flow

Lesson: Some obfuscation patterns are structural, not cosmetic. The obfuscator deliberately made the junk instructions affect program state.

14.5 Exhaustive Brute Force (Layer 8)

Approach: Test all $62^3 = 238,328$ alphanumeric 3-character combinations for Layer 8's final positions.

Why it failed:

- The solution (`_Ya}`) includes the closing brace as part of the hash computation
- Brute force tested patterns BEFORE the brace, not including it
- Off-by-one error in chunk boundary understanding

Lesson: Flag format delimiters (`{`, `}`) may participate in validation. Always consider that structural characters could be part of the cryptographic computation, not just formatting.

15. Conclusion

Flag: HTB{Sliced_Up_the_Function_4_Ya}

```
$ echo "HTB{Sliced_Up_the_Function_4_Ya}" | ./crackme
Correct. Validate the challenge using the flag: HTB{Sliced_Up_the_Function_4_Ya}
```

Key Methodological Insights

1. **Hardware watchpoints bypass code obfuscation** by monitoring data access rather than code flow.
When 48% of the binary is junk instructions designed to defeat static analysis, watching what the code *does* rather than *how* it's structured is the correct pivot.
2. **The deceptive accumulator is a psychological trap** designed to make analysts believe the challenge is unsolvable. The mathematical proof (constants OR to 0xffffffff) demonstrates the importance of rigorous analysis before drawing conclusions.
3. **Hidden conditional branches defeat CFG analysis** by embedding jumps inside apparent function calls.
Manual instruction-level tracing in GDB was required to discover the true decision point.

4. **Rolling state propagation** prevents middle-of-flag brute forcing. Each layer depends on all previous layers, requiring strictly sequential solution. This is the "Rubik's Cube" property: solving early layers wrong makes later layers impossible.
5. **Bit diffusion** at certain positions requires multi-character grid search rather than greedy single-character optimization. Recognizing when greedy fails and pivoting to grid search was critical for Layers 5 and 8.
6. **Delimiters may participate in validation**. The closing `}` is part of the Layer 8 hash, not a suffix. This insight—that the flag format itself is part of the cryptographic computation—was the final breakthrough.

Techniques Added to Library

- **Bitmask oracle attack**: Use per-character feedback from side-channel leakage for greedy solving
- **Hardware watchpoint pivot**: When code is too obfuscated, watch the data instead
- **Staircase layer detection**: Count checkpoint writes to map total validation layers
- **Zero-crossing search**: When residue approaches zero, scan adjacent ASCII systematically
- **Semantic pattern recognition**: Combine technical oracle data with linguistic analysis

Appendix A: Validation Architecture Diagram

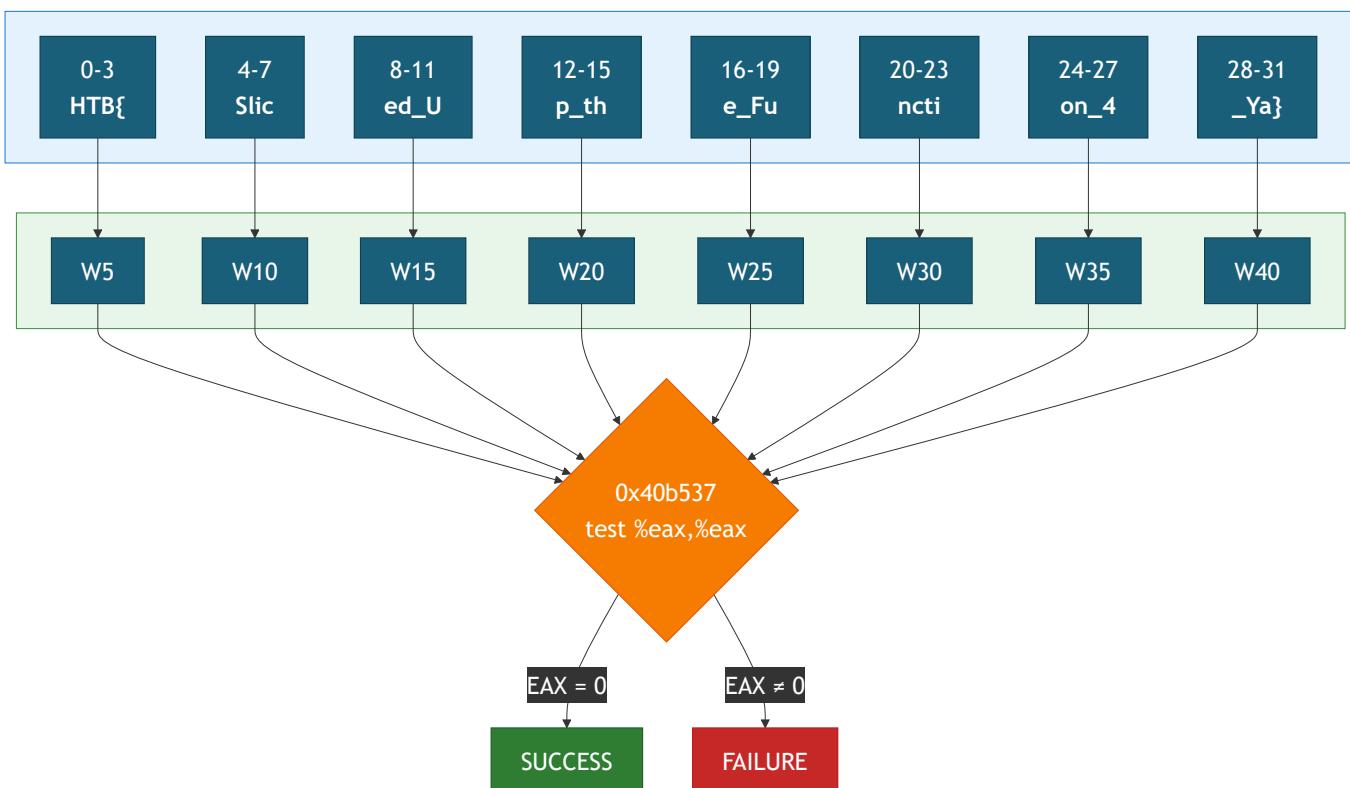


Figure 4: Complete validation architecture showing the 8-layer staircase validation with checkpoints and decision flow.

Key Memory Locations:

- Password Buffer: `0x40f080` (`scanf "%63s"`)
- Processing Buffer: `[rbp-0x18f0]` (dword array)
- Checkpoint Address: `0x40c4ef` (`mov %edx,-0x18f0(%rbp,%rax,4)`)

Decision Point:

- `0x40b537: test %eax,%eax` — Final decision: is accumulated result 0?
- `0x40bcc2: jne 0x4096e1` — Hidden jump to FAILURE if EAX ≠ 0
- `0x40bcc8: call 0x40b707` — Continue to SUCCESS if EAX = 0

Outcomes:

- SUCCESS: `0x40bdac → printf("Correct. Validate the challenge using flag: %s")`
 - FAILURE: `0x40b720 → puts("Incorrect flag. Try again")`
-

Appendix B: Analysis Session Summary

Session	Discovery	Impact
1-5	Obfuscation quantification, static analysis failures	Established baseline
6	Hardware watchpoint pivot, single validation point	Major breakthrough
7-10	Bitmask oracle, per-character solving	Found HTB{Slic
11-12	Deceptive accumulator mathematical proof	Avoided trap
13-14	Rubik's Cube dependency problem	Understood architecture
15-17	Staircase layer discovery, Layers 1-5 solved	HTB{Sliced_Up_the_Fu
18-19	Layers 6-7 solved, Layer 8 brute-force fails	HTB{Sliced_Up_the_Function_4
20	Layer 8 byte mapping, _4_ pattern	Near solution
21	Closing brace insight, semantic pattern testing	SOLVED

Key technique: Dynamic analysis via hardware watchpoints, pivoting away from static approaches