

FLARE-On 13 Challenge 4: ToxicMiner

By Ana Maria Martinez Gomez (@ana06@fosstodon.org)

Overview

ToxicMiner is a weaponized modification of the open-source **cpuminer-multi** CPU miner. **ToxicMiner** retains the appearance and functionality of a CPU miner, but it isn't mining Bitcoin—it is mining the environment! Instead of hashing blockchain data, the miner continuously hashes the local machine's Volume Serial Number (VSN) combined with a shifting nonce using a poisoned SHA-256 algorithm. If the miner finds a block matching a targeted Wallet Address, it triggers a decryption routine, using an intermediate state of that specific environment-keyed hash as an RC4 key to reveal the hidden flag:

cpu_m3lt3d_4nd_all_i_g0t_w4s_th1s_fl4g@flare-on.com.

- **Binary:** **cpuminer.exe**
- **SHA256:** **ee3a410dc86ea45a45c2aaedd071dd39b58035f066f8c9a134c93d4537a28a93**
- **Code:** <https://github.com/Ana06/toxic-miner>

Initial Inspection

When executed, the binary immediately outputs typical mining stats without attempting network connections, revealing itself as a modification of the open-source crypto miner **tpruvot/cpuminer-multi**:

```
** cpuminer-multi 1.3.7 by tpruvot@github **
BTC donation address: 1FhDPLPpw18X4srecguG3MxJYe4a1JsZnd (tpruvot)

[2026-07-03 07:54:30] 2 miner threads started, using 'sha256d' algorithm.
[2026-07-03 07:54:30] CPU #0: 8942 kH/s
[2026-07-03 07:54:30] CPU #1: 8386 kH/s
[2026-07-03 07:54:30] Total: 17328 kH/s
```

We examine the binary in IDA Pro, using the open-source code as a reference to rename key elements in our IDB such as importing the **algos** enum and labeling functions and globals. This helps us identify that standard mining networking and pool thread routines are present but entirely bypassed by hardcoded flags set right after the command-line parsing logic at address **0x140100330**.

```

GetSystemInfo(&SystemInfo);
g_num_cpus = SystemInfo.dwNumberOfProcessors;
if ( (int)SystemInfo.dwNumberOfProcessors < 1 )
    g_num_cpus = 1;
parse_cmdline((unsigned int)argc, argv);
g_opt_benchmark = True;           // Bypasses pool networking to run offline
g_opt_algo = ALGO_SHA256D;        // Locks the mining algorithm to SHA-256d (0x27)
if ( !g_opt_n_threads )

```

Figure 1: `opt_benchmark = 1 (True)`, `opt_algo = ALGO_SHA256D (0x27)`

You may notice the string `bTFuM195MHVyXzB3bl9idXMxbjNzcw==@flare.com` assigned to the `rpc_pass` variable. Decoding the Base64-encoded portion yields `m1n3_y0ur_0wn_bus1n3ss@flare.com`, an obvious decoy flag. Since the binary is hardcoded to run in offline benchmark mode, it never actually connects to a pool or uses this variable, making it clearly just a honeypot designed to trick regex-based AI solvers.

Identifying the Hashing Loop and the Poisoned Cryptography

Comparing our compiled binary's control flow against the upstream open-source codebase using static string references such as `"thread %d create failed"`, we locate the main thread worker function `miner_thread` being passed as an argument to `pthread_create` at `0x140100871`.

```

pthread_attr_init(thr + 24);
v53 = pthread_create(thr + 8, (_QWORD *)(thr + 24), (__int64)miner_thread, thr);
v54 = thr + 24;
error = v53;
sub_1401B24A0(v54);
if ( error )
{
    applog(0, "thread %d create failed", i);
    return 1;
}

```

Figure 2: Pseudocode view of `pthread_create` call at `0x14010087B`

The `miner_thread` function is responsible for spawning and managing the worker threads to their respective algorithm loops using a massive `switch(opt_algo)` block. By following the control flow down this switch-case dispatch table, we locate the case where the active mining algorithm is set to `ALGO_SHA256D (0x27)` and the call to the function `scanhash_sha256d` at `0x1401016DD`.

```

case ALGO_SHA256D:
    v42 = scanhash_sha256d(thr_id, (__int64)work, max_nonce, (__int64)&hashes_done);
    goto LABEL_219;
case ALGO_SIA:

```

Figure 3: Pseudocode view of call to `scanhash_sha256d` at `0x1401016DD`

Comparing the disassembly of `scanhash_sha256d` (at `0x140134F70`) against the upstream C code exposes a structural anomaly indicating this function has been significantly modified. The function now calls `GetVolumeInformationA` to obtain the VSN. The core hashing function (at `0x140134CA0`) is a poisoned SHA-256 implementation, which we can easily recognize by inspecting its constants: the first two standard

SHA-256 constants (`0x428a2f98` and `0x71374491`) have been swapped, and the total number of constants is halved to 32 entries.

```
.rdata:00000001401BD270 g_poisoned_k dd 71374491h, 428A2F98h, 0B5C0FBCFh, 0E9B5D8A5h, 3956C258h
.rdata:00000001401BD270 ; DATA XREF: sub_140135240+FF01o
.rdata:00000001401BD284 dd 59F111F1h, 923F82A4h, 0AB1C5ED5h, 0D807AA98h, 12835801h
.rdata:00000001401BD298 dd 243185BEh, 550C7DC3h, 72BE5D74h, 80DEB1FEh, 9BDC06A7h
.rdata:00000001401BD2AC dd 0C198F174h, 0E49B69C1h, 0EFBE4786h, 0FC19DC6h, 240CA1CCh
.rdata:00000001401BD2C0 dd 2DE92C6Fh, 4A7484AAh, 5CB0A9DCh, 76F988DAh, 983E5152h
.rdata:00000001401BD2D4 dd 0A831C66Dh, 0B00327C8h, 0BF597FC7h, 0C6E00BF3h, 0D5A79147h
.rdata:00000001401BD2E8 dd 6CA6351h, 14292967h
```

Figure 4: Global round constants table at `0x1401BD270`

By analyzing the surrounding logic of the success block, we can deduce the overall cryptographic architecture of the challenge. The binary is looking for a specific Volume Serial Number (VSN) and Nonce combination that, when processed by the poisoned SHA-256 function, produces a hash that matches the hardcoded target wallet address. However, the VSN and the Nonce play two completely separate roles. The VSN acts as the actual environmental decryption key, since the static RC4-encrypted flag is decrypted using `Poisoned_SHA256(VSN, 0x0B501E7E)` as the key (with the nonce fixed to `0x0B501E7E`). The Nonce, on the other hand, is purely a verification mechanism to satisfy the miner's native block-found check against the target wallet hash. Because the flag decryption key is completely independent of the nonce, a player can decouple the 2D search space into a simple 1D search. By forcing the nonce to `0x0B501E7E` and continuously decrypting the embedded ciphertext, they can verify if a candidate VSN is correct by checking if the decrypted output matches the standard, universal FLARE-On flag suffix: `@flare-on.com`.

Solving via Binary Patching

An easy and elegant way to solve the challenge without reversing the poisoned SHA-256 math is to patch the binary to brute-force the VSN keyspace. We can use IDA Pro to apply the following 4 patches:

- **1. Map Thread Nonce to VSN:** Natively, the threads sweep a local nonce and reuse the global VSN. We swap this instruction to force the threads to sweep the VSN instead.
 - Address: `0x1401351BE`
 - Original bytes: `8B 0D 84 D9 35 00` (`mov ecx, cs:VolumeSerialNumber`)
 - Patched bytes: `41 8B CE 90 90 90` (`mov ecx, r14d`)
- **2. Fast-Forward Jump:** We bypass the first validation execution and jump straight into the RC4 decryption block for every loop iteration.
 - Address: `0x140135090`
 - Original bytes: `0F 10 0D C9 81 08 00` (`movups xmm1, xmmword ptr [rip+0x881c9]`)
 - Patched bytes: `E9 29 01 00 00 90 90` (`jmp loc_1401351BE`)
- **3. Stack Pointer Fix:** In the unpatched binary, RC4 executes only once before exiting, safely reusing the stack space. Because our patch forces RC4 to execute millions of times in a loop, this buffer continuously overwrites the `hashes_done` tracking pointer and crashes the miner. We lower the RC4 state buffer down to a safely unused offset (`[rsp+0x40]`) to ensure the brute-force loop can run continuously without crashing the orchestrator.
 - Addresses `0x1401351F5` & `0x1401351FA`: Change the offset byte `0x50` to `0x40`.

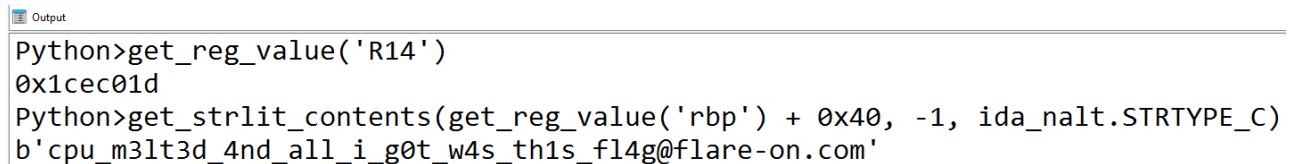
- **4. Static String Check & `int 3` Breakpoint:** At `0x140135200`, the binary naturally passes parameters to `applog` after decrypting. We know from the `rc4_decrypt` call at `0x1401351FB` that it explicitly decrypts exactly `0x33` (51) bytes. Since this 51-byte buffer starts at `[rbp+0x40]`, the target substring `@flare-o` is at `[rbp+0x66]`. We inject a check to verify this and trigger a breakpoint (`int 3`) when found.
 - Address `0x140135200`: Replace 23 bytes with:

```
48 B8 40 66 6C 61 72 65 2D 6F ; mov     rax, 'o-eralf@'
48 39 45 66                   ; cmp     [rbp+66h], rax
75 02                       ; jnz     short continue
CC                           ; int     3
90                           ; nop
E9 69 FE FF FF             ; jmp     next_candidate
```

The IDAPython script `ida_patcher.py` automates the patching.

Run the patched binary inside a debugger such as IDA Pro. Within less than 1 minute, the correct VSN will trigger the injected `int 3` instruction and pause execution. The `R14` register will hold the winning VSN: `0x01CEC01D`. We can extract the full decrypted flag from the local stack buffer (`[rbp+0x40]`) using IDAPython:

```
get_strlit_contents(get_reg_value('rbp') + 0x40, -1, STRTYPE_C)
```



```
Output
Python>get_reg_value('R14')
0x1cec01d
Python>get_strlit_contents(get_reg_value('rbp') + 0x40, -1, ida_nalt.STRTYPE_C)
b'cpu_m3lt3d_4nd_all_i_g0t_w4s_th1s_fl4g@flare-on.com'
```

Figure 5: Flag obtained with IDAPython

Obtained flag: `cpu_m3lt3d_4nd_all_i_g0t_w4s_th1s_fl4g@flare-on.com`.

Note on Debugger Execution: The patched binary was run with a single thread (`-t 1`) inside the IDA Debugger to prevent overhead and instability when tracking heavy, concurrent context-switching. Running the patched sample natively outside the debugger is technically faster and allows for stable multi-threading. However, because of the low VSN (`0x01CEC01D`), the single-threaded debugger approach is performant enough to brute-force the flag in less than 1 minute without leaving the IDA Debugger.