

crackmes.one CTF 2026 - crackme9

Feb 25, 2026

This year I participated in the crackmes.one CTF, a reverse engineering similar to Flare-On. Crackme9 was one of the harder challenges, with 43 solves by the end of the CTF.



As usual, we're given a prompt to enter a serial number. Tracing through the function that checks the serial, we can see a call to a function at the address `0x40a000`. This address is not part of the `.text` section of the PE file; instead, it occurs at the start of a nonstandard section called `.pc`. Looking at a hex dump of the `.pc` section, it was obviously encrypted.

```
99 ae 95 d7 80 31 f7 5c 3b c9 38 6e 9a 11 56 06 .....1.\;.8n..V.
16 81 d5 0c eb 59 b4 fa fe da 10 89 de 78 32 6b .....Y.....x2k
87 bb fb e6 09 19 77 a9 84 33 4a dc 91 7b 8b 5a .....w..3J..{.Z
6d 5c 4a 94 ed ee cd 5e d8 5e f5 c2 8b bc 35 15 m\J....^..^....5.
39 7c 1b 08 bd 76 29 5d 66 13 0c 14 c1 f3 93 c6 9|...v)]f.....
98 71 b1 26 83 bf c7 20 b9 a6 8b 91 cc 05 7f 20 .q.&... ..
b7 74 0c 6d ea 66 b3 34 aa 03 18 25 23 4f 45 26 .t.m.f.4...%#OE&
84 4a 2e 47 db 99 9d 97 6c 37 3a d3 84 ac 17 67 .J.G....17:....g
```

Initial deobfuscation steps

API resolution

I started off by setting breakpoints on commonly used APIs like `VirtualAlloc` and `VirtualProtect`. Tracing through the calls, I found that the APIs were being resolved through some kind of indirect lookup function.

```
0040112c    int32_t do_VirtualProtect()

0040112f        sub_401367()
0040113c        jump(sub_401ad3(&data_40d1a8))
```

As usual for this type of challenge, the APIs were being called based on their hash, in this case `0x10066f2f` for `VirtualProtect`.

```
00401ad3    int32_t __fastcall sub_401ad3(int32_t* arg1)

00401ae6        if (arg1[5] != 0)
00401aeb            return arg1[5]
00401aeb
00401b20        void* var_14
00401b20        arg1[5] = sub_401175(sub_401ca2(*arg1, &var_14, 0x10066f2f))
00401b26        return arg1[5]
```

Looking at the hash function at `sub_401ba9`, it turned out to just be unmodified CRC32:

```
00401ba9    uint32_t __stdcall sub_401ba9(char* arg1, int32_t arg2)

00401bad        int32_t esi = 0
00401baf        uint32_t edx = 0xffffffff
00401bb3        char* edi = arg1
00401bb3
00401bb9        if (arg2 u> 0)
00401bbb            int32_t ebx
00401bbb            int32_t var_10_1 = ebx
00401bbb
00401be6        do
00401bbc            ebx.b = edi[esi]
00401bc1            int32_t i_1 = 8
00401bdd            int32_t i
00401bdd
00401bdd        do
00401bc7            uint32_t ecx_2 = edx u>> 1
00401bc9            char eax_2 = ebx.b ^ edx.b
00401bcd            edx = ecx_2 ^ 0xedb88320
```

```

00401bcd
00401bd5          if ((eax_2 & 1) == 0)
00401bd5              edx = ecx_2
00401bd5
00401bd8          ebx.b u>>= 1
00401bda          i = i_1
00401bda          i_1 -= 1
00401bdd          while (i != 1)
00401bdf              edi = arg1
00401be2              esi += 1
00401be6          while (esi u< arg2)
00401be6
00401bf0          return not.d(edx)

```

Since CRC32 is such a widely used algorithm, we can use [HashDB](#) to look up the API names that go with each checksum. In fact, there's a [Binary Ninja plugin](#) that pulls data from HashDB to define an enum mapping all the API names to their checksums.

```

enum hashdb_strings_crc32 : uint32_t
{
    FlushInstructionCache = 0xe9258e7a,
    RtlMoveMemory = 0x1c0cd35c,
    GlobalUnWire = 0x2c82f5fb,
    MapUserPhysicalPages = 0xafd472a5,
    // [...]
    RtlUniform = 0xd09d867b,
    RtlNewSecurityObjectEx = 0x8733c5dd,
    WinSqmSetEscalationInfo = 0x648930f1
};

```

Normally I'd write a script to rename all the API hashing functions to their respective API names, but in this case there weren't that many of them so I just renamed them all by hand.

Breakpoints

When left to run under a debugger, the binary runs until it hits an int3 breakpoint at `0x4037c6`. I'd seen things like this before where exceptions are used to obfuscate control flow, but the binary seemed to be "getting stuck", never progressing past the breakpoint location.

```

004037c2      void __fastcall do_breakpoint(void* arg1)

```

```
004037c2  c6410101          mov     byte [ecx+0x1], 0x1
004037c6  cc               int3
```

When I was resolving the API hashes, I saw that

`KiUserExceptionDispatcher` was one of the hashed APIs, so I looked at how that was being used. It looked like the binary was supposed to overwrite the entry point of `KiUserExceptionDispatcher` with an exception handler of its own. However, for whatever reason the overwrite function wasn't being called, so the real `KiUserExceptionDispatcher` was being called instead.

```
00402ce0  void __fastcall overwrite_KiUserExceptionDispatcher(char* h

00402ce6          if (*hook == 0)
00402ce6          return

00402cef          int32_t KiUserExceptionDispatcher
00402cef          int80_t st0_1
00402cef          st0_1, KiUserExceptionDispatcher =
00402cef          get_KiUserExceptionDispatcher(&kernel32_addr, get_ke
00402cfd          _memcpy_s(KiUserExceptionDispatcher, 6, &hook[1], 6)
00402d05          *hook = 0
```

It turned out the `KiUserExceptionDispatcher` overwrite was only called if an anti-debug check was passed. The check used `NtQueryInformationProcess`, which I had thought could be bypassed with `ScyllaHide`, but for whatever reason that didn't work in this case. Ultimately, I just patched out the check entirely.

```
00402b6d  int32_t __fastcall check_ProcessDebugPort(int32_t* arg1)

00402b77          if (sub_402bc7(arg1) != 0)
00402b7c          return 1

00402b7c

00402b88          int32_t* ProcessInformation = arg1
00402b89          int32_t esi
00402b89          int32_t var_c = esi
00402b8a          ProcessInformation = nullptr
00402b91          get_dll_baseaddr()
00402b98          HANDLE ProcessHandle = do_GetCurrentProcess()
00402b9f          get_dll_baseaddr()
00402bb1          NTSTATUS result = do_NtQueryInformationProcess(ProcessHa
00402bb1          ProcessInformationClass: ProcessDebugPort, &ProcessI
00402bb1          ProcessInformationLength: 4, ReturnLength: nullptr)
00402bb1
```

```

00402bb9      if (result != STATUS_SUCCESS)
00402bc3          result.b = 0
00402bc6          return result
00402bc6
00402bbe          result.b = ProcessInformation != result
00402bc2      return result

```

At that point, the custom exception handler was being called, but the binary still crashed soon after with mysterious access violation exceptions. It seemed like it was trying to decrypt and run the `.pc` section, but the decryption was incorrect.

```

00402e1c      void decrypt_and_ntcontinue(PCONTEXT arg1) __noreturn

00402e20          context_record = arg1
00402e28          exception = __return_addr
00402e2d          get_cipher_ctx()
00402e40          decrypt_from_exception(ctx: &cipher_ctx, exception: exce
00402e40              context: context_record)
00402e45          get_dll_baseaddr()
00402e54          do_NtContinue(ContextRecord: context_record, TestAlert:
00402e59          breakpoint

```

The .pc section

Decrypting the code

At some point in the reversing process I got pretty stuck, and I was just kind of clicking around the decompilation for a while looking for something comprehensible. I eventually found something I recognized at `sub_401e67`:

```

00401e67      uint32_t* __thiscall sub_401e67(uint32_t* arg1, char* arg2,

00401e72          int32_t var_14 = 0xb979379e
00401e7c          int32_t var_10 = 0x157c4a7f
00401e83          int32_t var_c = 0x60c09cf3
00401e8a          int32_t var_8 = 0x34c8ed5c
00401e96          *arg1 = bytes_to_int(&var_14)
00401ea1          arg1[1] = bytes_to_int(&var_10)
00401ead          arg1[2] = bytes_to_int(&var_c)
00401ebd          arg1[3] = bytes_to_int(&var_8)
00401ec5          arg1[4] = bytes_to_int(arg2)

```

```

00401ed1     arg1[5] = bytes_to_int(&arg2[4])
00401edd     arg1[6] = bytes_to_int(&arg2[8])
00401ee9     arg1[7] = bytes_to_int(&arg2[0xc])
00401ef5     arg1[8] = bytes_to_int(&arg2[0x10])
00401f01     arg1[9] = bytes_to_int(&arg2[0x14])
00401f0d     arg1[0xa] = bytes_to_int(&arg2[0x18])
00401f1d     arg1[0xb] = bytes_to_int(&arg2[0x1c])
00401f20     arg1[0xc] = 0
00401f27     arg1[0xd] = 0
00401f33     arg1[0xe] = bytes_to_int(arg3)
00401f42     arg1[0xf] = bytes_to_int(&arg3[4])
00401f4a     return arg1

```

This function takes a 32-byte array and an 8-byte array, and uses them alongside a 16-byte constant to initialize an array of 64 bytes. If you've reversed enough cryptography code before, you'll likely recognize this as the initialization of a ChaCha20 matrix. Normally the constant used in ChaCha20 is the string `expand 32-byte k`, but I guess the challenge author thought that would be too easy to search for, so here it's replaced by the custom constant `0xb979379e`, `0x157c4a7f`, `0x60c09cf3`, `0x34c8ed5c`.

Looking at where the ChaCha20 matrix is initialized, the nonce is hard-coded as `0a 0b 0c 0d 0e 0f 10 11`, but the key is retrieved from a `get_key_from_hprov` function.

```

00404059     char* __thiscall do_chacha_with_hprov_key(struct cipher_ctx*

00404069         char* ciphertext_buf = do_malloc(ctx->len)
00404077         memcpy(dest: ciphertext_buf, src: ciphertext, count: ctx->len)
00404087         int32_t counter_1 = ciphertext - *ctx->pc_ptr
0040408c         int64_t nonce
0040408c         nonce.d = 0xd0c0b0a
00404093         nonce:4.d = 0x11100f0e
0040409a         get_crypt_context()
004040ad         struct matrix matrix
004040ad         init_chacha_matrix_with_counter(&matrix, key: get_key_from_hprov,
004040ad             counter0: 0, counter1: 0)
004040ba         chacha_rotated_counter(&matrix, counter_1, counter_2: 0)
004040c8         do_chacha_encrypt(&matrix, ciphertext: ciphertext_buf, len: ctx->len)
004040d3         return ciphertext_buf

```

I found that the key was being stored alongside an `HCRYPTPROV` handle obtained from `CryptAcquireContextA`, and that other Crypto API

functions were being called to generate the key itself.

```
00403388      BOOL __convention("regparm") hash_something_for_chacha_key(i

00403388          *0xff45c6 = arg1
00403395          arg2[-3] = 0
004033a5          get_dll_baseaddr()
004033aa          arg2[-5] = 0x40d1a0
004033b2          arg2[-4] = *arg2[-2]
004033c5          arg2[-5]
004033c8          BOOL result = do_CryptCreateHash(hProv: arg2[-4], Algid:
004033c8              dwFlags: 0, phHash: &arg2[-3])
004033c8
004033cf          if (result != 0)
004033de              get_dll_baseaddr()
004033e3              arg2[-9] = 0x40d1a0
004033e9              arg2[-8] = arg2[-3]
004033f7              arg2[-9]
004033f7
00403401              if (do_CryptHashData(hHash: arg2[-8], pbData: arg2[-
00403401                  dwFlags: 0) != 0)
0040340c                  arg2[-0xa] = 0x20
00403420                  get_dll_baseaddr()
00403425                  arg2[-0xc] = 0x40d1a0
0040342b                  arg2[-0xb] = arg2[-3]
00403440                  arg2[-0xc]
00403443                  do_CryptGetHashParam(arg2[-0xb], 2, arg2[-2] + 4
00403443
00403456                  get_dll_baseaddr()
0040345b                  arg2[-0xe] = 0x40d1a0
00403461                  arg2[-0xd] = arg2[-3]
00403467                  arg2[-0xe]
0040346a                  result = do_CryptDestroyHash(arg2[-0xd])
0040346a
00403473                  *arg2
00403474          return result
```

Setting a breakpoint on `CryptHashData`, it turned out the ChaCha20 key was the SHA-256 hash of the `.text` section of the executable. This is an anti-debug measure, as setting software breakpoints changes the contents of `.text`.

When I tried dumping the `.text` section from the binary and hashing it, the resulting hash didn't result in a valid decryption. Looking more closely in x64dbg at the data that was being hashed, I found

that the last byte of the section had been changed from `0x00` to `0x90`. Making the same change to my copy of the `.text` section, I got the correct key `F630AA38D57297375D645559C334FD50D55CA1D177D2655A042351CF69244BF2`. (For the actual decryption step, I used [CryptoTester](#), which supports custom-constant ChaCha20 out of the box.)

Running the code

Single stepping

Now that I had the right key, I could see how the program tried to run the decrypted code. The program obtains the ChaCha20 key with a call to `CryptGetHashParam`, so I set a breakpoint on `CryptGetHashParam`'s return to patch in the correct hash. To make things easier, I eventually figured out that I could set the breakpoint to run a command to write to the hash address, so that I wouldn't have to manually do the patch on every run. The syntax to do this in x64dbg is:

```
set 0x40d268,#F630AA38D57297375D645559C334FD50D55CA1D177D2655A042351CF69
```

However, even with the correct key, the program didn't decrypt and run the `.pc` section all at once. Instead, it only decrypted it 16 bytes at a time, and wrote only those few instructions to the buffer it had allocated for running shellcode.

```
004040ad      init_chacha_matrix_with_counter(&matrix, key: get_key_fr
004040ad      counter0: 0, counter1: 0)
004040ba      chacha_rotated_counter(&matrix, counter_1, counter_2: 0)
004040c8      do_chacha_encrypt(&matrix, ciphertext: ciphertext_buf, 1
004040d3      return ciphertext_buf
```

Moreover, the program seemed to only be running one instruction at a time before allocating a new shellcode buffer with `VirtualAlloc`. For example, out of the first 16 bytes that are decrypted, only the instruction `mov eax,ecx` is run at first.

```
3E4C0006 | 8BC1          | mov eax,ecx
3E4C0008 | C741 04 90B16E0A | mov dword ptr ds:[ecx+4],A6EB190
3E4C000F | C700 00000000    | mov dword ptr ds:[eax],0
3E4C0015 | 0000          | add byte ptr ds:[eax],al
```


3E4C0017	0000	add byte ptr ds:[eax],al
3E4C0019	0000	add byte ptr ds:[eax],al

Then, a new buffer is allocated to run the next instruction `mov dword ptr ds:[ecx+4],A6EB190`, and so on.

17100002	C741 04 90B16E0A	mov dword ptr ds:[ecx+4],A6EB190
17100009	C741 08 336C4720	mov dword ptr ds:[ecx+8],20476C33
17100010	0000	add byte ptr ds:[eax],al
17100012	0000	add byte ptr ds:[eax],al
17100014	0000	add byte ptr ds:[eax],al

In order to run a single instruction at a time before stopping execution, the program uses the same mechanisms that debuggers use to perform single steps. The CPU has a flag called the *trap flag* that tells it to execute a single instruction before raising an exception of type `EXCEPTION_SINGLE_STEP`. (You can see an example of how a debugger might use this flag [here](#).) Sure enough, the trap flag is set before each run of the decrypted shellcode.

```
004037fa    uint32_t __stdcall set_context_flags(CONTEXT* arg1)

00403804        *(arg1->ContextFlags + 0x18) = 0
00403809        *(arg1->ContextFlags + 0x14) = 0
0040380e        *(arg1->ContextFlags + 4) = 0
00403811        uint32_t ContextFlags = arg1->ContextFlags
00403813        *(ContextFlags + 0xc0) |= 0x100
0040381e        return ContextFlags
```

Control flow obfuscation

Even after decryption, the decompilation of the `.pc` section didn't make much sense. The code was full of `int3` breakpoints, which were usually followed by nonsensical disassembly.

```
0040a0cb  mov     eax, dword [ebp-0x8]
0040a0ce  and     eax, 0x80000003
0040a0d3  int3
0040a0d4  fisttp  dword [eax-0x7d], st0
0040a0d7  enter   0x40fc, 0x85
0040a0db  ror     ah, 0x38
```

Upon closer inspection in x64dbg, several bytes were being skipped each time the program executed an `int3` breakpoint. For instance,

after the breakpoint at `0x40a035` was executed, the next instruction to be run was the `push dword ptr [ebp+8]` instruction at `0x40a037`, with the address `0x40a036` being skipped over entirely.

Looking at the exception handler code again, there were different functions for retrieving the next `.pc` instruction depending on whether the exception was an `int3` breakpoint or a single step.

```
00404562    void __thiscall decrypt_from_exception(struct cipher_ctx* ctx,
00404569        if (ctx->field_8 != 0)
00404569            return
00404569
00404574        if (*exception == STATUS_BREAKPOINT)
00404579            chacha_breakpoint(ctx, context)
00404574        else if (*exception == STATUS_SINGLE_STEP)
0040458b            chacha_single_step(ctx, context)
00404586        else if (*exception == STATUS_GUARD_PAGE_VIOLATION)
0040459a            CONTEXT* context_1 = context
0040459e            chacha_guard_page_violation(ctx, exception)
```

Unlike the single-step handler, which just set `context->Eip` to the next instruction before returning, the breakpoint handler called another function to determine the next instruction pointer before returning.

```
004042ec    int32_t __thiscall chacha_breakpoint(struct cipher_ctx* ctx,
004042fe        if (sub_4037b2(ctx->field_24) == 0)
00404318            CONTEXT* exception_context = context
00404318
00404328            if (check_context_eip_in_range(ctx, eip: exception_c
0040432d                breakpoint_set_eip(ctx, exception_context)
00404332                CONTEXT* context_1 = context
00404335                uint32_t Eip = context_1->Eip
0040433b                uint32_t Eip_1 = Eip
0040433f                ctx->Eip = Eip
00404342                chacha_and_flush_cache(ctx, context: context_1)
004042fe        else
00404304            flush_instruction_cache(ctx, &context)
00404309            ctx->field_24
00404310            set_context_flags(&context)
00404310
0040434d        return 0xffffffff
```

Looking at the `breakpoint_set_eip` function at `0x4045f3`, it was retrieving a value from a map, then setting the next instruction to one of two locations pulled from that map.

```
004045f3    struct map_val* __thiscall breakpoint_set_eip(struct cipher_
004045f7        CONTEXT* exception_context_1 = exception_context
00404608    struct map_val* result =
00404608        get_map_data(ctx, key: (exception_context_1->Eip).w
00404608
00404611    if (result != 0)
00404622        int32_t offset
00404622
00404622        if (check_eflags(result, exception_context_1) == 0)
0040462c            offset = result->offset_2
00404622    else
00404627        offset = result->offset_1 + result->offset_2
00404627
0040463c        flush_process_cache(ctx, &exception_context, offset
00404641        result.b = 1
00404611    else
00404613        result.b = 0
00404613
00404647    return result
```

Dumping the map data from the debugger, it contained a long list of structures of four integer values. The first integer was the map key, corresponding to the address of the `.pc` instruction currently being executed (minus the base address of `0x400000`). The third and fourth integers were the two choices of addresses to jump to.

```
00 A0 00 00 01 00 00 00 2F 00 00 00 02 00 00 00
01 A0 00 00 12 00 00 00 49 00 00 00 02 00 00 00
02 A0 00 00 0A 00 00 00 72 00 00 00 02 00 00 00
03 A0 00 00 0A 00 00 00 43 00 00 00 02 00 00 00
04 A0 00 00 05 00 00 00 5A 00 00 00 02 00 00 00
```

The second integer, which was always a value between 1 and 18, was used for deciding which of the two offsets to jump to. Depending on its value, different combinations of the flags in the `eflags` register were checked, and the decision for where to jump was made based on the value of those flags.

```

00404166      uint32_t __stdcall check_eflags(struct map_val* arg1, CONTEXT
00404172          uint32_t EFlags = arg2->EFlags
00404178      uint32_t choice = arg1->field_4 - 1
00404178
0040417c      if (choice > 0x11)
00404238          choice.b = 0
0040417c      else
00404182          switch (choice)
004041ea              case 0
004041ea                  EFlags u>>= 6 // ~ZF
004041ed                  goto get_negated_flag
00404234              case 1
00404234                  choice.b = 1
00404189              case 2
0040418c                  EFlags.b = (EFlags u>> 7).b & 1 // SF
0040418f                  choice.b = EFlags.b
0040418f
0040418f      // [...]
0040418f
004041a2          case 0x10
004041a5              if ((1 & (EFlags u>> 6).b) != 0) // ZF
004041b7                  choice.b = 0
004041b7
004041a5              else
004041af                  // ~ (OF ^ SF)
004041af                  choice.b = (EFlags u>> 0xb).b ^ (EFlags
004041af
004041b3                  if ((1 & choice.b) == 0)
004041b7                      choice.b = 1
004041b7
004041b3                  else
004041b7                      choice.b = 0
004041b7
0040418c          case 0x11
0040418c              EFlags.b &= 1
0040418f              choice.b = EFlags.b
0040418f
0040418f
0040423b      return choice

```

The combinations of flags that were being checked were, in fact, the exact combinations of flags that are checked in ordinary x86 branch instructions. For example, in the switch case above, case 0 corresponds to the condition `ZF = 0`, which is used in the conditional branch instruction `JNZ`. Similarly, case 0x10 corresponds to the condition `ZF = 0` and `SF = OF`, which is the conditional `JG`.

Essentially, every single branch instruction in the `.pc` section had been replaced with a breakpoint, and the branch logic was being handled in the exception handler. Since this is a one-to-one replacement of one assembly instruction with another, the most logical way to deal with it would've been to patch the original branch instructions back in, except that the branch instructions were longer than the breakpoint instructions replacing them.

Reversing the check algorithm

Emulating with Unicorn

Since I couldn't think of a good way to patch the binary in a way that would get me a reasonable-looking decompilation, I decided to try to emulate the `.pc` section in Unicorn, adding a hook to intercept the breakpoint instructions and resolve the branches. I had to hook a couple of external functions like `strlen`, but luckily there weren't too many of those, so I didn't have to do much extra work other than the usual boilerplate that comes with setting up Unicorn.

```
from unicorn import *
from unicorn.x86_const import *
from capstone import *
import struct

def to_int(b):
    return hex(int.from_bytes(b, 'little'))

jumps = { 0: 'jnz', 1: 'jmp', 2: 'js', 3: 'jae', 4: 'jle', 5: 'ja', 6: 'jbe' }

def get_bit(val, bit):
    return (val >> bit) & 1

def get_branch_cond(cond, flags, ecx):
    if cond == 0: # jnz
        return get_bit(flags, 6) == 0
    elif cond == 1: # jmp
        return True
    elif cond == 2: # js
        return get_bit(flags, 7) != 0
    elif cond == 3: # jae
        return get_bit(flags, 0) == 0
    elif cond == 4: # jle
```

```

        return get_bit(flags, 6) == 0 or (get_bit(flags, 0xb) != get_bit
elif cond == 5: # ja
    return get_bit(flags, 6) == 0 and get_bit(flags, 0) == 0
elif cond == 6: # jge
    return get_bit(flags, 7) == get_bit(flags, 0xb)
elif cond == 7: # jp
    return get_bit(flags, 2) != 0
elif cond == 8: # jo
    return get_bit(flags, 0xb) != 0
elif(cond == 9): # jbe
    return get_bit(flags, 6) != 0 or get_bit(flags, 0) != 0
elif(cond == 10): # jecxz
    return ecx == 0
elif(cond == 11): # jnp
    return get_bit(flags, 2) == 0
elif(cond == 12): # jz
    return get_bit(flags, 6) != 0
elif(cond == 13): # jl
    return get_bit(flags, 0xb) != get_bit(flags, 0x7)
elif(cond == 14): # jns
    return get_bit(flags, 7) == 0
elif(cond == 15): # jno
    return get_bit(flags, 0xb) == 0
elif(cond == 16): # jg
    return get_bit(flags, 6) == 0 and (get_bit(flags, 0xb) == get_bi
elif(cond == 17): # jb
    return get_bit(flags, 0) != 0
else:
    raise ValueError("invalid condition", cond)

```

```
offsets = {}
```

```
f = open('data.bin', 'rb').read()
```

```
for i in range(0, len(f), 0x10):
```

```
    data = f[i:i+0x10]
```

```
    key, cond, jump, base = struct.unpack('<4i', data)
```

```
    offsets[key + 0x400000] = (cond, jump, base)
```

```
def emu_push(mu: Uc, val):
```

```
    esp = mu.reg_read(UC_X86_REG_ESP)
```

```
    mu.mem_write(esp - 4, val.to_bytes(4, 'little'))
```

```
    mu.reg_write(UC_X86_REG_ESP, esp - 4)
```

```
def emu_pop(mu: Uc):
```

```
    esp = mu.reg_read(UC_X86_REG_ESP)
```

```
    val = mu.mem_read(esp, 4)
```

```
    mu.reg_write(UC_X86_REG_ESP, esp + 4)
```

```

    return int.from_bytes(val, 'little')

def emu_return(mu: Uc):
    ret_addr = emu_pop(mu)
    mu.reg_write(UC_X86_REG_EIP, ret_addr)

def hook_strlen(uc):
    print('Skipping strlen function')
    uc.reg_write(UC_X86_REG_EAX, 0x13)

def hook_crypt(uc):
    #print('Skipping crypto setup')
    uc.reg_write(UC_X86_REG_EAX, HPROV)

def hook_hash(uc):
    #print('Skipping hash function')
    uc.reg_write(UC_X86_REG_EAX, MEM)

def hook_pass(uc):
    pass

def hook_mem_write(mu: Uc, access, address, size, value, user_data):
    data_bytes = value.to_bytes(size, 'little')
    print(f'[{hex(mu.reg_read(UC_X86_REG_EIP))}] Address {hex(address)}')

def hook_mem_read(mu: Uc, access, address, size, value, user_data):
    if address >= MEM and address < MEM + 0x10:
        data_bytes = mu.mem_read(address, size)
        print(f'[{hex(mu.reg_read(UC_X86_REG_EIP))}] Address {hex(address)}')

def hook_code(mu: Uc, address, size, user_data):
    for i in md.disasm(mu.mem_read(address, size), address):
        print(f"0x%x:\t%s\t%s" % (i.address, i.mnemonic, i.op_str))
        print(f'\teax:', hex(mu.reg_read(UC_X86_REG_EAX)))
        print(f'\tebx:', hex(mu.reg_read(UC_X86_REG_EBX)))
        print(f'\tecx:', hex(mu.reg_read(UC_X86_REG_ECX)))
        print(f'\tedx:', hex(mu.reg_read(UC_X86_REG_EDX)))
        ebp = mu.reg_read(UC_X86_REG_EBP)
        esp = mu.reg_read(UC_X86_REG_ESP)
        print(f'\tebp: {to_int(mu.mem_read(ebp, 4))}   ebp - 4: {to_int(mu.mem_read(ebp - 4, 4))}')
        print(f'\tstack: +0: {to_int(mu.mem_read(esp, 4))}   +4: {to_int(mu.mem_read(esp + 4, 4))}')

        if i.mnemonic == 'call': # skip external function calls
            target = i.operands[0].imm
            if target in hooks:
                hooks[target](mu)

```

```

        mu.reg_write(UC_X86_REG_EIP, i.address + i.size)

def hook_interrupt(mu: Uc, intno, user_data):
    if intno == 3:
        eip = mu.reg_read(UC_X86_REG_EIP) - 1
        cond, jump, base = offsets[eip]
        not_taken_addr = eip + base
        taken_addr = eip + base + jump
        #print(hex(eip), hex(not_taken_addr), hex(taken_addr))

        ecx = mu.reg_read(UC_X86_REG_ECX)
        flags = mu.reg_read(UC_X86_REG_EFLAGS)

        should_branch = get_branch_cond(cond - 1, flags, ecx)
        if should_branch:
            print(f'Branch {jumps[cond - 1]} taken: {hex(taken_addr)}')
            mu.reg_write(UC_X86_REG_EIP, taken_addr)
        else:
            print(f'Branch {jumps[cond - 1]} not taken: {hex(not_taken_a
            mu.reg_write(UC_X86_REG_EIP, not_taken_addr)

sc = open('shellcode.bin', 'rb').read()

md = Cs(CS_ARCH_X86, CS_MODE_32)
md.detail = True

hooks = {
    0x409731: hook_strlen,
    0x403477: hook_crypt,
    0x40a4b5: hook_hash
}

BASE = 0x40a000
BASE_SIZE = 1024*1024
STACK_ADDR = 0x40000000
STACK_SIZE = 1024*1024

MEM = 0x60000000
MEM_SIZE = 1024*1024

HPROV = MEM + 0x100
CONSTS = MEM + 0x2000

def emu(start):
    mu = Uc(UC_ARCH_X86, UC_MODE_32)

```



```

mu.mem_map(STACK_ADDR, STACK_SIZE)
mu.reg_write(UC_X86_REG_ESP, STACK_ADDR + STACK_SIZE // 2)
mu.reg_write(UC_X86_REG_EBP, STACK_ADDR + STACK_SIZE // 2 - 0x1000)

mu.mem_map(MEM, MEM_SIZE)
mu.mem_write(MEM, b'ABCD-4567-89ab-cdef')
mu.mem_write(STACK_ADDR + STACK_SIZE // 2 + 4, MEM.to_bytes(4, 'little'))

KEY = bytes.fromhex('F630AA38D57297375D645559C334FD50D55CA1D177D2655')
mu.mem_write(HPROV + 4, KEY)
mu.mem_write(HPROV + 0x24, (MEM+0x1000).to_bytes(4, 'little'))
mu.mem_write(HPROV + 0x28, b'\x00\x8a\x00\x00')

mu.mem_write(CONSTS, bytes.fromhex('47bb5d8690b16e0a336c472093768a1c'))
mu.reg_write(UC_X86_REG_ECX, CONSTS)

mu.mem_map(BASE, BASE_SIZE)
mu.mem_write(BASE, sc)

mu.hook_add(UC_HOOK_CODE, hook_code)
mu.hook_add(UC_HOOK_INTR, hook_interrupt)

mu.hook_add(UC_HOOK_MEM_READ, hook_mem_read)

mu.emu_start(start, BASE+BASE_SIZE)

emu(0x40a025)

```

Unfortunately, this approach turned out to be largely unhelpful. I could tell that the serial was supposed to be 19 characters long (consistent with something of the format `XXXX-XXXX-XXXX-XXXX`), and that the characters were being hashed in groups of four, but the hash algorithm didn't match up with anything I recognized.

I did notice that the `.pc` section started with a function that loaded five random-looking values, though, so I figured those were the target values for the hash.

```

0040a000      int32_t* __convention("fastcall") sub_40a000(int32_t* arg1)

0040a000  c70147bb5d86      mov     dword [ecx], 0x865dbb47 {0x865dbb47, 0x865dbb47, 0x865dbb47, 0x865dbb47}
0040a006  8bc1              mov     eax, ecx
0040a008  c7410490b16e0a    mov     dword [ecx+0x4], 0xa6eb190
0040a00f  c74108336c4720    mov     dword [ecx+0x8], 0x20476c33
0040a016  c7410c93768a1c    mov     dword [ecx+0xc], 0x1c8a7693

```

```
0040a01d c74110fbbdfe59 mov dword [ecx+0x10], 0x59febdfb
0040a024 c3               retn    {__return_addr}
```

Fixing the decompilation

After a while I gave up on emulation and came back to the idea of trying to decompile the code again. I couldn't figure out a way to patch in the branch instructions without overwriting too many other instructions, so instead I decided to see if I could modify the

Binary Ninja

In Binary Ninja, the code to be decompiled is lifted through several different forms of IL as it's being processed. In theory, there are APIs to edit IL instructions at any stage of the analysis process. Since the modification I wanted to make was supposed to be equivalent to a simple binary patch, I thought it would make the most sense to modify the lowest level of IL, called *Lifted IL*. However, when I looked through the [documentation](#) on how to do so, I found that it explicitly advised against it:

You probably do not want to modify Lifted IL with a Workflow. Instead, consider modifying the Architecture directly (most of which are Open Source on our GitHub) or making your Activity modify Low Level IL.

"Modifying the Architecture directly" sounded promising, so I looked at how the [x86 architecture plugin](#) generated IL instructions.

The [code](#) for how breakpoints were handled turned out to be very straightforward: for each INT3 breakpoint instruction, a single IL breakpoint instruction is emitted. To fix it, I would need to modify that line to emit an IL branch instruction instead.

```
case XED_ICLASS_INT3:
    il.AddInstruction(il.Breakpoint());
    break;
```

Conveniently, the [code](#) for handling branch instructions is right next to the breakpoint code, and it includes a convenient switch case listing all the ways to construct an IL conditional branch from its corresponding x86 branch.

```
case XED_ICLASS_JO:
    ConditionalJump(arch, il, il.FlagCondition(LLFC_O), addrSize, br
```

```

        return false;

    case XED_ICLASS_JNO:
        ConditionalJump(arch, il, il.FlagCondition(LLFC_NO), addrSize, b
        return false;

// [...]

    case XED_ICLASS_JECXZ:
        ConditionalJump(arch, il, il.CompareEqual(4, il.Register(4, XED_
        return false;

    case XED_ICLASS_JRCXZ:
        ConditionalJump(arch, il, il.CompareEqual(8, il.Register(8, XED_
        return false;

```

To generate the branch instructions, I started by adding a hard-coded map of all the breakpoint addresses in the `.pc` section and their corresponding branch types, as well as an enum containing all the conditions.

```

#include <map>
#include <cstdint>

struct CrackmeCond {
    int cond;
    int not_taken;
    int taken;
};

const uint64_t CRACKME_SC_START = 0x40a000;
const uint64_t CRACKME_SC_END = 0x40a5ff;

enum class JumpCond: int {
    JNZ = 0,
    JMP,
    JS,
    JAE,
    JLE,
    JA,
    JGE,
    JP,
    JO,
    JBE,
    JECXZ,
    JNP,

```

```

    JZ,
    JL,
    JNS,
    JNO,
    JG,
    JB
};

std::map<int, CrackmeCond> conds = {
    { 0x40a000, { 1, 2, 49 } },
    { 0x40a001, { 18, 2, 75 } },
    { 0x40a002, { 10, 2, 116 } },
    { 0x40a003, { 10, 2, 69 } },
    { 0x40a004, { 5, 2, 92 } },
    { 0x40a005, { 3, 2, 34 } },
    { 0x40a006, { 18, 2, 161 } },
    // [...]
    { 0x40a5fd, { 7, 2, 130 } },
    { 0x40a5fe, { 13, 2, 52 } },
    { 0x40a5ff, { 13, 2, 46 } },
};

```

I then modified the breakpoint handling code to add a switch case modeled after Binary Ninja's own code for handling x86 conditional branch instructions, calling `ConditionalJump` with the arguments I had parsed out of the map whenever a breakpoint fell within the range of addresses in the `.pc` section.

```

case XED_ICLASS_INT3:
    if(addr >= CRACKME_SC_START && addr < CRACKME_SC_END)
    {
        const auto cond_struct = conds[addr];
        const auto cond = static_cast<JumpCond>(cond_struct.cond);
        const auto taken_addr = addr + cond_struct.taken;
        const auto not_taken_addr = addr + cond_struct.not_taken;

        switch(cond)
        {
            case JumpCond::JMP:
                il.AddInstruction(il.Jump(il.ConstPointer{not_taken_addr}));
                break;

            case JumpCond::JO:
                ConditionalJump(arch, il, il.FlagConditionalJump{cond});
                break;

```

```

// [...]

case JumpCond::JECXZ:
    ConditionalJump(arch, il, il.CompareEqual);
    break;

default:
    il.AddInstruction(il.Breakpoint());
}

}

else
{
    il.AddInstruction(il.Breakpoint());
}

break;

```

After making this modification, the LLIL breakpoint instructions were replaced with LLIL branches, which repaired the control flow of the binary.

```

6 @ 0040a02e  [ebp - 0x14 {var_18}].d = ecx
7 @ 0040a031  [ebp + 8 {serial}].d
8 @ 0040a035  breakpoint

```

```

6 @ 0040a02e  [ebp - 0x14 {var_18}].d = ecx
7 @ 0040a035  if ([ebp + 8 {serial}].d == 0) then 8 @ 0x40a045 else 10 @

```

This worked far better than I expected it to, resulting in coherent decompilation! It wasn't perfect by any means, but it was good enough for me to reimplement the algorithm. The trace I'd printed out from Unicorn turned out to be helpful after all, as it gave me values to test against in the steps where the decompilation was ambiguous.

```

0040a025      int32_t __stdcall do_check(int32_t arg1 @ ecx, int32_t serial

0040a025      int32_t ebp
0040a025      int32_t var_4 = ebp
0040a026      struct struct_1* ebp_1 = &var_4
0040a02b      void* entry_ebx
0040a02b      void* var_28 = entry_ebx
0040a02c      int32_t entry_esi
0040a02c      int32_t var_2c = entry_esi

```

```

0040a02d      int32_t edi
0040a02d      int32_t var_30 = edi
0040a02d      int32_t* esp_1 = &var_30
0040a02e      int32_t var_18 = arg1
0040a035      int32_t result

0040a035
0040a035      if (serial == 0)
0040a045          result.b = 0
0040a035      else
0040a03f          esp_1 = &var_30
0040a03f
0040a043          if (0x409731(serial) == 0x13)
0040a04c              int32_t var_8_1 = 0
0040a053              int32_t var_c_1 = 0
0040a05a              int32_t var_14_1 = 0
0040a061              int32_t var_10_1 = 0xcafebabe
0040a061
0040a06f          while (true)
0040a06f              sub_40a4b5(0x403477())
0040a06f
0040a080              if (ebp_1->serial_char == 0)
0040a08a                  ebp_1->serial_char = 1
0040a080              else if (ebp_1->serial_char == 1)
0040a0ae                  *(esp_1 - 4) = zx.d(*(ebp_1->acc + ebp_1
0040a0af                  *(esp_1 - 8) = ebp_1->i
0040a0ba                  ebp_1->i = do_char_hash(ebp_1->field_c)
0040a0c1                  ebp_1->count += 1
0040a0ce                  int32_t done_yet = ebp_1->count & 0x8000
0040a0ce
0040a0d3                  if (done_yet < 0)
0040a0d9                      done_yet = ((done_yet - 1) | 0xffff
0040a0d9
0040a0dc                  if (done_yet == 0)
0040a0e6                      // go to compare-and-multiply
0040a0e6                      ebp_1->serial_char = 2
0040a0dc                  else if (ebp_1->count != 0x13)
0040a10f                      // if not all chars hashed, resume I
0040a10f                      ebp_1->serial_char = 1
0040a0f3                  else
0040a0fe                      ebp_1->serial_char = 3
0040a09a                  else if (ebp_1->serial_char != 2) // state
0040a178                      if (ebp_1->serial_char != 3)
0040a178                          break
0040a178
0040a17a                      *(esp_1 - 4) = 4
0040a186                      ebp_1->field_0 = ebp_1->field_c[*(esp_1

```

```

0040a199      int32_t check_result = ebp_1->field_10 |
0040a19b      ebp_1->field_10 = check_result
0040a19b
0040a19e      if (check_result != 0)
0040a1a9          ebp_1->serial_char = 5  // wrong
0040a19e      else
0040a1a0          ebp_1->serial_char = 4  // right
0040a11f      else // state 2: get compare val and multip
0040a124      void* eax_11
0040a124      int32_t edx_1
0040a124      edx_1:eax_11 = sx.q(ebp_1->count)
0040a12a      unimplemented {sar eax, 0x2}
0040a12e      ebp_1->field_8 = ((eax_11 + (edx_1 & 3))
0040a13a      // save compare_val into ebp_1 - 0x1c?
0040a13a      ebp_1->field_4 = *(ebp_1->field_c + (ebp
0040a151      ebp_1->field_10 |= ebp_1->i ^ ebp_1->fie
0040a168      ebp_1->i = ebp_1->count * 0x112233 - 0x3
0040a16b      ebp_1->serial_char = 1
0040a16b
0040a1b6      if (ebp_1->serial_char != 4)
0040a1be          result.b = 0
0040a1b6      else
0040a1b8          result.b = 1
0040a043      else
0040a045          result.b = 0
0040a045
0040a1c7      *esp_1
0040a1c7      esp_1[1]
0040a1c8      esp_1[2]
0040a1ca      ebp_1->field_20
0040a1cb      return result

```

Reimplementation

The hash was a weird ad-hoc algorithm that made different decisions about how to process each character depending on certain ranges of values that it fell into. This means that the initial approach I took with Unicorn would never have worked very well on its own, as the flow of execution is completely different depending on whether each of the characters in the serial are letters, numbers, special characters and so on. There's even a special case that specifically handles the character `0x30`.

The presence of all the weird conditionals also means that the hash can't be cracked using constraint solvers. (My initial

reimplementation of the algorithm was in Python with the intent of solving it with z3, which resulted in confusing error statements like `Symbolic expressions cannot be cast to concrete Boolean values.`) Luckily, only 4 characters are hashed at a time, so I rewrote my implementation in C and bruteforced each group of characters.

```
#include <stdio.h>
#include <stdint.h>

uint32_t hash_round(uint32_t acc, uint32_t c) {
    if((acc & 1) != 0) {
        if((acc ^ c) <= 0x80000000) {
            if((c >= 0x61) && (c <= (0x61 + 26))) {
                acc -= c;
            }
            else if((c < 0x41) || (c > (0x41 + 26))) {
                acc *= 9;
            }
            else {
                acc += c;
                if((acc & 0x100) != 0) {
                    acc ^= 0x13371337;
                }
            }
        }
        else if(c > 0x60) {
            acc = (c * 0x21) ^ (acc + 0xdeadbeef);
        }
        else {
            acc += 0xdeadbeef;
        }
    }
    else {
        if(c >= 0x40) {
            if(c % 2 != 0) {
                acc = ((acc >> 5) | (acc << 0x1b)) ^ 0x87654321;
            }
            else {
                acc = ((acc << 3) | (acc >> 0x1d)) + 0x12345678;
            }
        }
        else if((c & 2) == 0) {
            acc -= (c << 4);
            if(c == 0x30) {
                acc |= 0xf0f0f0f0;
            }
        }
    }
}
```



```

    }
    else {
        acc ^= 0x55aa55aa;
        acc += c;
    }
}

uint32_t mod_5 = (c % 5) + 2;
for(uint32_t i = 0; i < mod_5; i++) {
    if((acc & 0x80000000) == 0) {
        acc <<= 1;
    }
    else {
        acc = (acc << 1) ^ 0x4c11db7;
    }

    uint32_t mask = i & 0x80000001;
    if(mask != 0) {
        acc += i * 0xa;
    }
    else {
        acc ^= c;
    }
}

uint32_t count = acc;
count ^= (count >> 0x10);
count ^= (count >> 0x8);

if((count & 0xf) > 7) {
    acc = ~acc;
}

return acc;
}

uint32_t hash(char* data, uint32_t pos, uint32_t len) {
    uint32_t acc = 0xcafebabe + 0x112233 * 4 * pos;
    for(uint32_t i = 0; i < len; i++) {
        acc = hash_round(acc, data[i]);
    }

    return acc;
}

int check_val(uint32_t expected, uint32_t pos, uint32_t len) {

```

```

for(char i = 0x20; i < 0x7f; i++) {
    //printf("trying %x\n", i);
    for(char j = 0x20; j < 0x7f; j++) {
        for(char k = 0x20; k < 0x7f; k++) {
            for(char l = 0x20; l < 0x7f; l++) {
                char to_try[4] = {i, j, k, l};
                uint32_t acc = hash(to_try, pos, len);
                if(acc == expected) {
                    printf("%c%c%c%c", i, j, k, l);
                    return 0;
                }
            }
        }
    }
}

return -1;
}

int main() {
    uint32_t compares[5] = {0x865dbb47, 0xa6eb190, 0x20476c33, 0x1c8a769, 0x1c8a769};

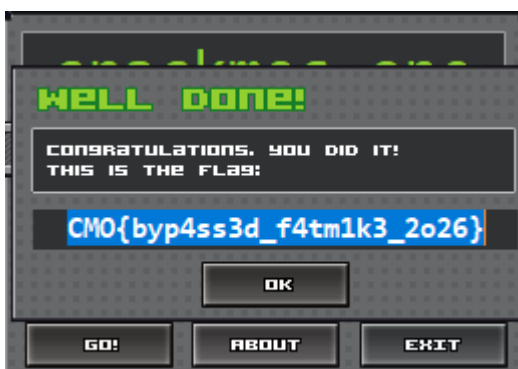
    for(int i = 0; i < 4; i++) {
        if(check_val(compares[i], i, 4) == -1) {
            return -1;
        }
    }

    if(check_val(compares[4], 4, 3) == -1) {
        return -1;
    }

    return 0;
}

```

This gets us the serial: D3FE-A7ED-BAAD-C0D3. Entering it into the crackme, we finally have our flag!



```
CMO{byp4ss3d_f4tmlk3_2o26}
```

Note: After the CTF, I found out that the control flow obfuscation came from an obfuscator called [Nanomites](#), the source code of which was on the challenge author's GitHub long before the CTF started. I'll have to remember to check for that next time.

./amnesia.sh

```
./amnesia.sh
```



[clairelevin](#)

Malware analyst, reverse engineer, and occasional CTFer.



[claire-levin-a8b50b256](#)