Nederlands Forensisch Instituut
Ministerie van Justitie en Veiligheid

Create your own hashcat plugin

Pelle Kuiters & Ewald Snel

Contents

Welcome to the workshop

This workshop will teach you how to add a simple plugin to hashcat. At the end of the day, you will have a working plugin *and* know what steps to follow to add it to hashcat.

The purpose of this workshop is to understand how hashcat works and what you need to do, more or less, to create your own plugin. After the workshop, you will have the knowledge to start working on your own plugins.

We will not explain in detail how hashcat works during the workshop. Want to learn more? Run into any problems? The Hashcat Plugin Development Guide contains a lot of information. The manual can be found in the docs folder of the hashcat repository.

For this workshop, you need to possess basic knowledge of security and cryptography. It is also important that you know how to use hashcat via the command line. And that you understand how to enter the correct parameters to carry out the desired attack.

1. Hashcat knowledge boost

Hashcat is a powerful open-source tool to crack passwords by using the GPU and sometimes CPU as well. In very simple attacks, hashcat is able to crack billions of password hashes per second.

During an attack, hashcat generates and processes password candidates and compares each result against the hash you are trying to crack. If it matches, the attack is successful and the password has been found.

This first chapter contains information that is useful before you start the workshop. After reading it, you will have learned important terminology and understand the best way to create a plugin. If you're familiar with all of this already, that's even better. You can move on to chapter 2 straight away.

Example of a successful MD5 attack

```
8743b52063cd84097a65d1633f5c74f5:hashcat

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 8743b52063cd84097a65d1633f5c74f5
Time.Started.....: Tue Jul 11 09:28:21 2023 (0 secs)
Time.Estimated...: Tue Jul 11 09:28:21 2023 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: hashca?l [7]
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:    79172 H/s (0.03ms) @ Accel:2048 Loops:1 Thr:32 Vec:1
Speed.#*.........:    79172 H/s
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 26/26 (100.00%)
Rejected.........: 0/26 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: hashcan -> hashcaq
Hardware.Mon.#1..: Temp: 29c Fan: 30% Util:  0% Core:1725MHz Mem:7000MHz Bus:16
```

Want to know more about how to use hashcat? Check out the wiki at https://hashcat.net/wiki/, where you can find frequently asked questions and more.

1.1 Programming in hashcat

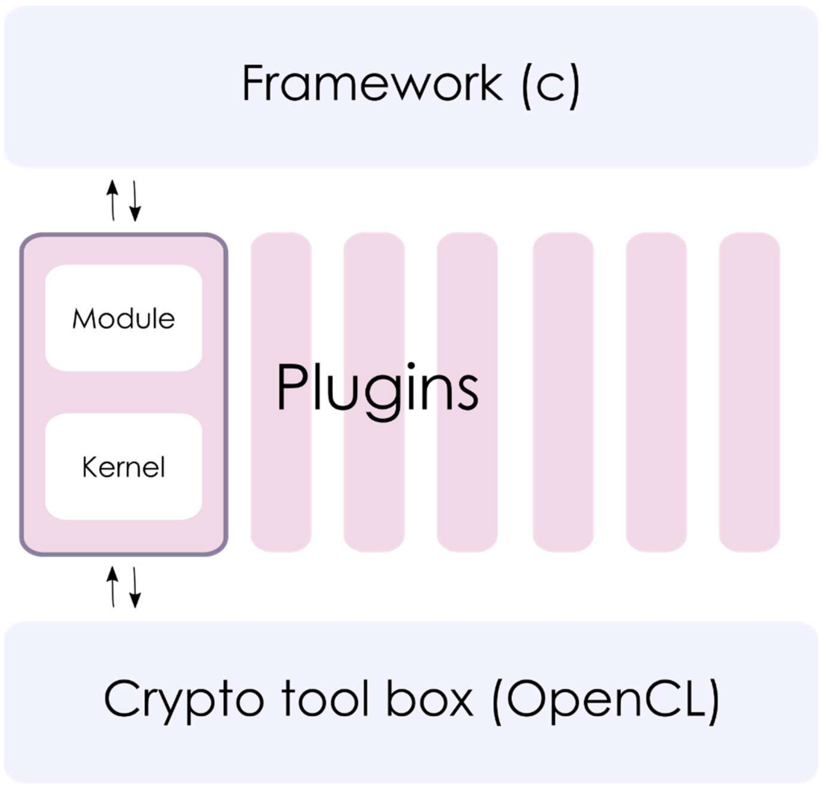
The hashcat framework is written in C and the crypto-algorithms are written in OpenCL for the GPU.

You need two components to add a plugin to hashcat: a module and a kernel:

- The module contains the attack's *configuration* written in C. This allows you to benefit optimally from hashcat's features and structure.
- The kernel contains the cryptographic operations that allow you to go from password to answer. This is written in OpenCL so it can be executed on the GPU.

Very convenient! You do not have to know how to control the GPU and copy data. Hashcat will handle all of it for you. All you have to do is focus on the cryptographic part.

Relationship between the different parts of hashcat



1.3 Inside and outside kernels

Hashcat supports two kernel types: inside and outside. The kernel file name indicates which attack the hash mode is able to carry out and the kernel type.

m<number>-pure.cl	– If it contains the attack number without any other information, it is an 'outside' kernel.
m<number>_a3-pure.cl **m<number>_a3-optimized.cl**	- Does the file name contain a0, a1 or a3? Then it is an 'inside' kernel for a specific attack mode.

Inside kernel

The 'inside' kernel is intended for fast algorithms where generating the passwords may take up more time than executing the cryptographic operations.

Outside kernel

The 'outside' kernel is used for slow attacks where executing the cryptographic operations takes up most of the time, for instance when hashcat has to execute a large number of iterations or when a lot of memory is used. The advantage of the outside kernel is that you only have to create a single kernel file.

1.4 Three attack modes

Hashcat has different attack modes that you can use. These modes are wordlist, combinator and brute-force. You need to understand these three modes to create your own plugin:

1. Wordlist [-a 0]
   Hashcat tries all of the words on a list. For instance, one compiled based on context information from a case. This type of attack is also called a dictionary attack.
2. Combinator [-a 1]
   Hashcat combines two word lists.
3. Brute-force [-a 3]
   Instead of trying existing words, you can also choose to generate new combinations. You can enter the following option in the console, for instance: '`hello?d?d?d?d?d`'. This means that hashcat tries all combinations from '`hello00000`' to '`hello99999`' in the attack.

Not every attack has the same performance. For instance, if an attack cracks 10 passwords per second, it makes more sense to compile a dictionary containing likely password candidates then using the brute-force option.

The OpenCL folder contains the kernels for all the hash modes. The file name shows what type of attack the hash mode covers:

m00050_a0-pure.cl **m00050_a0-optimized.cl**	– **50** stands for the hash mode ‘HMAC-MD5 (key = $pass)’. – **a0** shows that it is a wordlist attack. Enter –a 0 in the command prompt if you want to start a wordlist attack. This is therefore an *inside* kernel.
m00050_a1-pure.cl **m00050_a1-optimized.cl**	– **50** stands for the hash mode ‘HMAC-MD5 (key = $pass)’. – **a1** shows that this is a combinator attack.
m00050_a3-pure.cl **m00050_a3-optimized.cl**	– **50** stands for the hash mode ‘HMAC-MD5 (key = $pass)’. – **a3** shows that this is a brute-force attack.
m00400-pure.cl	– 400 stands for the hash mode ‘phpass’ – This hash mode is suitable for all attack types. You can tell because there is no a0, a1 or a3 in the file name. This is therefore an *outside* kernel.

Difference between pure and optimized

Some kernel files are *pure* while others are *optimized*, but what is the difference?

- The 'pure' attack can process password lengths of up to 256 characters. Hashcat expects your plugin to be able to handle this maximum number of characters. Want to process fewer characters, such as for a 40-character password? Then you must state this explicitly in your module.
- The 'optimized' attack is optimized for performance and can usually process fewer characters. The maximum password length is printed on the console when you start the attack. Generally, the maximum password length is 32 characters for optimized kernels. Want to try short passwords only? Then we recommend creating an optimized kernel since its performance is usually better.

1.5 GPU *and* CPU

Hashcat is optimized for using the GPU for computations in an attack. The GPU is powerful and can handle multiple complex computations at the same time. However a powerful CPU is also important these days. Anti-GPU algorithms are more and more common, such as Scrypt and Argon. These algorithms require more memory and less traditional computations.

It is important to consider, in each attack, what cryptographic operations are required and what hardware or combination of hardware is most suitable.

If you want to start working on a new attack, check whether hashcat already contains a similar attack first. You can use that for your own attack. If there is no similar attack, opt for an outside kernel. Is the algorithm very fast? Then you can still convert your attack to an inside kernel.

No stress, we will tell you more about this during the workshop.

1.6 Sharing a plugin

This workshop teaches you how to develop a new plugin. The advantage of a plugin is that it only contains the code that is needed for *your* attack. This way you don't have to change the source code of Hashcat. If you want to share your plugin, you only need to share your module and kernel file. And if there is a new hashcat release that breaks your plugin, you only need to edit the plugin.

2. To do *before* the workshop

It is important that you prepare some things on your laptop or computer prior to the workshop. This chapter explains what to do *and* why it is necessary.

You need a laptop with Linux or macOS for the workshop. If you are using a Windows laptop, you need to install the WSL (Windows Subsystem for Linux). You also need an IDE or text editor you feel comfortable working with.

We recommend using version control. Git is a good option. You do not need it for the workshop but it will come in handy once you start working on your own plugins. You can use git to try out changes and revert to a stable point if necessary.

You must install Hashcat on your laptop for the workshop, of course. We explain the necessary steps for this below.

Step 1 ➥ Unpack

This workshop contains the following files:

- This manual
- A sample hash ➥ example.hash
- Ready-made module and kernel: module_12345.c and m12345-pure.cl. Useful for having a look if you run into any trouble. Once you complete the workshop, you will have made your own files.
- Blank module and kernel: module_template.c and m-template-pure.cl. You will fill these templates during the workshop.

Step 2 ➥ Install and compile hashcat

1. Install Hashcat v6.2.6 on your laptop. You can find this version on the hashcat website.
2. In the console, navigate to the /hashcat folder and run: `make clean; make`

Step 3 ➥ a quick test

Want to know for sure that the compilation process was successful? Do this quick test.

- Try to crack this hash: `fc3c55df519b1fe37d132bccfc788d15`
- Use this command: `./hashcat -a 3 -m 0 <hash> ?l?l?ldiebol`

What commands are you giving to hashcat with this?

- The -a stands for attack mode and 3 stands for brute-force
- The -m stands for hash-type and 0 for MD5
- The hash you want to crack is in <hash>
- The ? stands for charset and the l for lowercase
- The letters 'diebol' mean that the password ends with these characters

So you are searching for a password that consists of 9 lowercase letters: studiebol

This command uses an algorithm included in hashcat by default. Are you curious about the other options? Use **`--help`**.

Step 4 ➡ Create your own reference implementation

If you are going to create a plugin, it is inconvenient to start working directly in hashcat. After all, you do not want to have to focus on hashcat's configuration before knowing what your hash looks like. It is also important to understand every step of the algorithm before starting in hashcat, so it is advisable to create a reference implementation first. This allows you to flesh out all the properties of your attack beforehand.

A few tips:

- Create the reference implementation in a language you have mastered, such as Java or Python.
- Make sure you understand every step.
- Use print lines in every step so you can see whether the output is correct.

It is important that you follow this step and understand it thoroughly. It will provide the necessary details to create a plugin. This is important in crypto because the smallest mistake can lead to a hash not cracking. You also need this knowledge to execute optimizations.

Good to know: for the workshop, we will be using a (made up) cryptographic implementation that uses MD5 to hash the password and AES-128 to subsequently decrypt the data.

In this implementation, MD5 is used as key derivation function (KDF). This turns a password into the key for the next step. The next step consists of using AES-128 to decrypt some data using the key. Usually you need an initial vector (IV) when using AES. However, to keep the example simple, we opted for using an IV that consists entirely of zeroes.

Create your own reference implementation for the workshop

Have enough time and feel like it? Create your own reference implementation that executes the steps below. The encrypted data you will decrypt with your reference implementation is shown below.

Input for your reference implementation

```
Salt           : dc1153d3069912c5ce8c4fc75797c5b4
Iterations     : 10000
Encrypted data : 49493e63559ba3669b1b14ad958aaed1086433a1bcd2d1ac9df28ced1d14f810
```

```
Sample input hash
dc1153d3069912c5ce8c4fc75797c5b4:10000:49493e63559ba3669b1b14ad958aaed1086433a1bcd2d1ac
9df28ced1d14f810
```

You need to carry out a few steps to decrypt the encrypted data.

1. Run the password ‘hashcat’ and salt in the table through MD5.

Output

```
Password
    Hex                                                 | Text
   -----------------------------------------------------+-------------------
    68 61 73 68 63 61 74                                | hashcat

Salt
    Hex                                                 | Text
   -----------------------------------------------------+-------------------
    DC 11 53 D3 06 99 12 C5 CE 8C 4F C7 57 97 C5 B4     | ..S.......O.W...

Result of MD5 round 1:
    Hex                                                 | Text
   -----------------------------------------------------+-------------------
    CF BD BF 9C B5 49 74 C4 7F FB CF 9E ED 5B EA 74     | .....It......[.t
```

2. Replace the first four bytes with **0**, **1**, **2**, **3**

Output

```
MD5 hash first 4 bytes replaced with 0x00, 0x01, 0x02, 0x03:
    Hex                                                 | Text
   -----------------------------------------------------+-------------------
    00 01 02 03 B5 49 74 C4 7F FB CF 9E ED 5B EA 74     | .....It......[.t
```

3. Now run this hash10.000 times through MD5

Output after 10,000 MD5 iterations

```
MD5 Digest after 10.000 iterations:
    Hex                                                 | Text
   -----------------------------------------------------+-------------------
    9F 64 87 2F 52 74 B0 E1 A7 AA 59 CE 05 01 2D 0B     | .d./Rt....Y...-.
```

4. Use this hash as a key for AES-128 decryption. For this implementations, we are using 16 zero-bytes as IV.

Decryption output

```
Encrypted Text:
    Hex                                              | Text
   --------------------------------------------------+------------------
    49 49 3E 63 55 9B A3 66 9B 1B 14 AD 95 8A AE D1  | II>cU..f........
    08 64 33 A1 BC D2 D1 AC 9D F2 8C ED 1D 14 F8 10  | .d3.............

Decrypted Text:
    Hex                                              | Text
   --------------------------------------------------+------------------
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
```

5. Check the decrypted data. See all zeroes? Congratulations, you have found the password.

If your output is incorrect, check the intermediate steps once again to determine where the problem lies.

Java reference implementation

Might be helpful and serve as inspiration for your own reference implementation.

Java reference example Workshop

```java
public class WorkshopReference {
    private static final byte[] IV = new byte[16]; // filled with 16 zero bytes
    private static final byte[] TARGET = new byte[32]; // filled with 32 zero bytes

    public static void main(final String[] args) throws Exception {
        // Password that is being used
        final byte[] passwordBytes = "hashcat".getBytes();
        // Salt of 16 bytes
        final byte[] salt = Utils.hexToBytes("dc1153d3069912c5ce8c4fc75797c5b4");
        // Number of rounds to perform the MD5
        final int rounds = 10000;
        // Encrypted data that we want to decrypt
        final byte[] encrypted =
Utils.hexToBytes("49493e63559ba3669b1b14ad958aaed1086433a1bcd2d1ac9df28ced1d14f810");

        // Calculate the MD5 of the password and salt
        final MessageDigest md5 = MessageDigest.getInstance("MD5");
        md5.update(passwordBytes, 0, passwordBytes.length);
        md5.update(salt, 0, salt.length);

        final byte[] initialDigest = md5.digest();

        // Replace the first 4 bytes with 0x00010203
        initialDigest[0] = 0;
        initialDigest[1] = 1;
        initialDigest[2] = 2;
        initialDigest[3] = 3;

        // Perform MD5 for the number of rounds specified
        byte[] digest = initialDigest;
        for (int i = 0; i < rounds; i++) {
            md5.update(digest, 0, digest.length);
            digest = md5.digest();
        }

        // Use the generated MD5 hash as a AES-key to decrypt the encrypted data
        // The IV is 16 zero bytes
        final byte[] aeskey = digest;

        final Cipher wrappingCipher = Cipher.getInstance("AES/CBC/NoPadding");
        final SecretKeySpec keySpec = new SecretKeySpec(aeskey, "AES");
        wrappingCipher.init(DECRYPT_MODE, keySpec, new IvParameterSpec(IV));

        final byte[] decryptedText = wrappingCipher.doFinal(encrypted);

        // Verify that the decrypted data only contains zeroes for success
        if (Arrays.equals(TARGET, decryptedText)) {
            System.out.println("SUCCESS!");
        }
        else {
            System.out.println("FAILED!");
        }
    }

}
```

3. The module

Did your reference implementation work? And you understand all the necessary steps? Then you are ready to start the workshop. This chapter describes all the necessary steps. We have prepared a number of things for you. This prevents you from getting stuck and ensures that you have a working plugin at the end of the workshop.

You need to create a module and a kernel for a plugin. The workshop contains two templates you can use to turn your reference implementation into a hashcat plugin. These templates contain the minimum basic elements for a plugin.

The module **`module_template.c`**	The module is written in C. In the module, you will specify how the input is processed before being sent to the GPU. The input is usually a hash.
The kernel **`m-template-pure.cl`**	The kernel is written in OpenCL. The kernel contains the cryptographic operations that generate the answer for a password candidate.

Step 1 ➥ Give your plugin a unique attack number

This number provides a unique identifier for your plugin. Choose a five digit number. This number ensures hashcat recognizes your plugin and uses it to start your attack.

Rename the templates:

1. Change `module_template.c` to `module_<number>.c`
2. Change `m-template-pure.cl` to `m<number>-pure.cl`

This number is not just used in the file names, it is also used in the code. Are you copying a module and kernel from another developer so you can modify them? Make sure to replace this number everywhere or you may get confusing error messages and your plugin might not work.

Step 2 ➥ Configure the module

1. Put the **module_<number>.c** file in the **`src/modules`** `folder`. Note: This also contains the module files for the other plugins.

2. Complete the configuration in your module. You can use the sample file **module_12345.c** as an example.

Below is an explanation of what you need to enter and where.

Basic configuration of the module template

```
static const u32   ATTACK_EXEC    = ATTACK_EXEC_OUTSIDE_KERNEL;
static const u32   DGST_POS0      = 0;
static const u32   DGST_POS1      = 1;
static const u32   DGST_POS2      = 2;
static const u32   DGST_POS3      = 3;
static const u32   DGST_SIZE      = DGST_SIZE_4_4;
static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH;
static const char *HASH_NAME      = <<"enter a brief description of your attack here">>;
static const u64   KERN_TYPE      = <<"enter a number for your attack here">>;
static const u32   OPTI_TYPE      = OPTI_TYPE_ZERO_BYTE;
static const u64   OPTS_TYPE      = OPTS_TYPE_PT_GENERATE_LE;
static const u32   SALT_TYPE      = SALT_TYPE_EMBEDDED;
static const char *ST_PASS        = <<"enter your test password here">>;
static const char *ST_HASH        = <<"enter your test hash here">>;
```

Explanation

`static const u32 ATTACK_EXEC = ATTACK_EXEC_OUTSIDE_KERNEL;`

This means that the plugin is one of the relatively slow attack modes. You can read more about this option in **1.4 Outside and inside kernels**.

`DGST_POS0, DGST_POS1, DGST_POS2, DGST_POS3`

These options indicate the order in which you want to verify the hash. Some encryption algorithms have the bytes in a different order after the last step for optimization purposes. In that case, you must specify the order. The order is not relevant for the plugin you are about to create since we will be checking for zeroes.

`static const u32 DGST_SIZE = DGST_SIZE_4_4;`

This is the size of the hash we want to verify. In our example, we wish to verify the final 16 bytes for zeroes, so we have opted for 4x4 digest size.

DGST stands for 'digest' and is often used for hash functions and cryptography.

```
static const u32 HASH_CATEGORY = HASH_CATEGORY_RAW_HASH;
```

You must specify the category for each attack. Refer to the –help to see what categories are available and determine which one is best suited for your attack.

```
static const char *HASH_NAME = <<"enter a brief description of your attack
here">>;
```

This should contain a short description of your attack. This is important so other developers will understand the purpose of your attack.

```
static const u64 KERN_TYPE = <<enter the attack number here>>;
```

Enter the unique number you chose for your plugin here. **Please note:** This number must be the same everywhere or your plugin will not work.

```
OPTS_TYPE_PT_GENERATE_LE
```

For this plugin, we selected outside kernel. Passwords on the outside kernel are always generated in little-endian. This is why it says GENERATE_LE.

Please note: For the SHA engine, the password must be provided in big-endian (BE), so in that case you would need to swap the password on the GPU. In general, you should determine in advance how the password should be provided.

```
SALT_TYPE_EMBEDDED
```

You will decrypt a piece of data with the plugin. This option indicates that you will use 'your own' data structure. This is not a standard 'salt', hashcat has a generic salt structure for this. For example, this struct can contain a master key or encrypted data. In other words, something that you cannot handle easily in the generic digest or salt structure.

```
static const char *ST_PASS = <<"enter your test password here">>;
```

This is the self-test hash, which comes in handy during development. It is customary to use the password 'hashcat' here.

Hashcat first always carries out a self-test. A failed self-test forces hashcat to run just once so you can test your own code quickly. The self-test is always followed by autotuning. This can generate a lot of noise on the console, making it difficult to track down your own print lines.

static const char *ST_HASH = <<"enter your test hash here">>;

Enter the hash for the test password 'hashcat' here. You can find this hash in the file 'example.hash'.

Example module_12345.c

Basic configuration in the module_12345.c

```
static const u32   ATTACK_EXEC    = ATTACK_EXEC_OUTSIDE_KERNEL;
static const u32   DGST_POS0      = 0;
static const u32   DGST_POS1      = 1;
static const u32   DGST_POS2      = 2;
static const u32   DGST_POS3      = 3;
static const u32   DGST_SIZE      = DGST_SIZE_4_4;
static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH;
static const char *HASH_NAME      = "Workshop Example";
static const u64   KERN_TYPE      = 12345;
static const u32   OPTI_TYPE      = OPTI_TYPE_ZERO_BYTE;
static const u64   OPTS_TYPE      = OPTS_TYPE_PT_GENERATE_LE;
static const u32   SALT_TYPE      = SALT_TYPE_EMBEDDED;
static const char *ST_PASS        = "hashcat";
static const char *ST_HASH        =
"dc1153d3069912c5ce8c4fc75797c5b4:10000:49493e63559ba3669b1b14ad958aaed1086433a1bcd2d1a
c9df28ced1d14f810";
```

Step 3 ➥ Define the struct

- Define your own struct
- Specify how much memory the struct requires
- Add the method to module_init()

Example struct from module_12345.c

```
typedef struct workshop
{
  u32 encrypted_data[8];

} workshop_t;

u64 module_esalt_size (MAYBE_UNUSED const hashconfig_t *hashconfig, MAYBE_UNUSED
const user_options_t *user_options, MAYBE_UNUSED const user_options_extra_t
*user_options_extra)
{
  const u64 esalt_size = (const u64) sizeof (workshop_t);

  return esalt_size;
}
```

Explanation

We have to define a struct which we can access in the module and in the kernel. It must contain 32 bytes of the encrypted data so we can decrypt that on the GPU. Above is the example from module_12345.c.

In the configuration, we used SALT_TYPE_EMBEDDED to specify that we will be using such a struct. The example illustrates how to define it. Since an u32 consists of 4 bytes and we have 32 bytes of encrypted data in all, we will need a total of 8 x u32.

In addition to defining the struct, you must also specify the size of the struct. This ensures that hashcat allocates the required amount of memory. You do this by entering the name of your struct in the method module_esalt_size(), in this case ‘workshop_t’. This is also shown in the example above.

Example init_module workshop module_12345.c

```
Is:
module_ctx->module_esalt_size               = MODULE_DEFAULT;

Becomes:
module_ctx->module_esalt_size               = module_esalt_size;
```

Hashcat must also know that the module_esalt_size() method is being used. You do this by specifying the module_esalt_size method in module_init(). By default, it is set to MODULE_DEFAULT and you will change it to module_esalt_size.

Step 4 ➥ Define the temporary struct

- Define the temporary struct
- Specify how much memory the struct needs
- Add the method to module_init()

Example struct tmps module_12345.c

```
typedef struct workshop_temp
{
  u32 digest_buf[4];

} workshop_temp_t;

u64 module_tmp_size (MAYBE_UNUSED const hashconfig_t *hashconfig,
MAYBE_UNUSED const user_options_t *user_options, MAYBE_UNUSED const
user_options_extra_t *user_options_extra)
{
  const u64 tmp_size = (const u64) sizeof (workshop_temp_t);

  return tmp_size;
}

void module_init (module_ctx_t *module_ctx){
  module_ctx->module_tmp_size                 = module_tmp_size;
}
```

Explanation

The temporary struct (tmps_t) ensures that hashcat maintains the data on the GPU and also processes it on the GPU. You will have to carry out an MD5 10,000 times for the workshop. We use this struct to store the intermediate results.

The temporary struct consists of 4xu32 because an MD5 hash consists of 16 bytes, so we need 4 x u32. We also specify the size of the struct and ensure that the method is specified in module_init() so hashcat knows that this struct is the one being used.

Step 5 ➥ Add the hash_decode() method

1. Add the hash_decode() method. In this method, the input is parsed by the tokenizer and the salt and encrypted data are placed in the correct structure.

The method has two parts: the tokenizer and the subsequent processing of tokens:

1. Tokenizer example from module_12345.c

```
int module_hash_decode (MAYBE_UNUSED const hashconfig_t *hashconfig, MAYBE_UNUSED
void *digest_buf, MAYBE_UNUSED salt_t *salt, MAYBE_UNUSED void *esalt_buf,
MAYBE_UNUSED void *hook_salt_buf, MAYBE_UNUSED hashinfo_t *hash_info, const char
*line_buf, MAYBE_UNUSED const int line_len)
{
  u32 *digest = (u32 *) digest_buf;

  workshop_t *workshop = (workshop_t *) esalt_buf;

  hc_token_t token;

  token.token_cnt = 3;

  // Salt
  token.len_min[0] = 32;
  token.len_max[0] = 32;
  token.sep[0]     = ':';
  token.attr[0]    = TOKEN_ATTR_VERIFY_LENGTH
                   | TOKEN_ATTR_VERIFY_HEX;

  // Rounds
  token.len_min[1] = 5;
  token.len_max[1] = 5;
  token.sep[1]     = ':';
token.attr[1]    = TOKEN_ATTR_VERIFY_LENGTH
                 | TOKEN_ATTR_VERIFY_DIGIT;

// Encrypted data
token.len_min[2] = 64;
token.len_max[2] = 64;
token.sep[2]     = ':';
token.attr[2]    = TOKEN_ATTR_VERIFY_LENGTH
                 | TOKEN_ATTR_VERIFY_HEX;

const int rc_tokenizer = input_tokenizer ((const u8 *) line_buf, line_len, &token);

if (rc_tokenizer != PARSER_OK) return (rc_tokenizer);

}
```

Explanation

3 elements

The tokenizer for the module consists of 3 elements:

- The salt
- The number of iterations
- The encrypted data

Number of characters

Every token shows the minimum and maximum length of the element. Our salt consists of 16 bytes, which amounts to 32 hexadecimal characters. The encrypted data consists of 32 bytes which amounts to 64 hexadecimal characters. The attribute ‘TOKEN_ATTR_VERIFY_HEX’ states that these two elements must be interpreted as hexadecimal.

Number of iterations

Your plugin should run the hash 10,000 times through MD5. To accomplish this the number of iterations, ‘10,000’, must be interpreted as decimal. We use the attribute ‘TOKEN_ATTR_VERIFY_DIGIT’ to achieve this.

Verification

In the last step, the tokenizer checks that the hash complies with these conditions. If your salt exceeds 32 characters, or the hash consists of more than 3 elements, the system will generate an error message.

2. Processing of tokenizer example from module_12345.c

```
// Salt information
const u8 *salt_pos = token.buf[0];
salt->salt_buf[0] = hex_to_u32(salt_pos + 0);
salt->salt_buf[1] = hex_to_u32(salt_pos + 8);
salt->salt_buf[2] = hex_to_u32(salt_pos + 16);
salt->salt_buf[3] = hex_to_u32(salt_pos + 24);
salt->salt_len = 16;

// Number of rounds - salt-iter determines the number of times the loop() is called
const u8 *rounds_pos = token.buf[1];
salt->salt_iter = hc_strtoul ((const char *) rounds_pos, NULL, 10);

// Encrypted data
const u8 *encrypted_data_pos = token.buf[2];
for(int index = 0; index < 8; index++)
{
  workshop->encrypted_data[index] = hex_to_u32(encrypted_data_pos + (index * 8));
}

// We are using this to create a unique digest
digest[0] = workshop->encrypted_data[0];
digest[1] = workshop->encrypted_data[1];
digest[2] = workshop->encrypted_data[2];
digest[3] = workshop->encrypted_data[3];

return (PARSER_OK);
```

Explanation

- To process the elements, you refer to a specific token buffer and then specify how to process it. In the example above, this translates to hex_to_u32() and hc_strtoul().
- The three elements are converted from hex characters to bytes so hashcat can work with them. These are stored in the corresponding fields of the structure for subsequent use.

Please note: Hashcat works primarily with u32. This is 4-bytes together!

Hashcat offers many tools to convert all kinds of input. We urge you to look for examples in other modules. For instance, if you need base64 or want to know how to import binary data.

Step 6 ➥ Add the hash_encode() method

1. Add the hash_encode() method, which ensures that the result is converted to the correct hash format. Hashcat then compares the converted result against hashes already in the potfile.

This is the example of this method from module_12345.c. The data in the different structures is converted back to the original hash format.

hash_module_encode () example from module_12345.c

```
 const u32 *digest = (const u32 *) digest_buf;

const workshop_t *workshop = (const workshop_t *) esalt_buf;

/* 64 hex characters - 32 bytes - encdata */
u8 encrypted_data[32];
memcpy (encrypted_data, workshop->encrypted_data, sizeof(workshop->encrypted_data));
u8 hex_encrypted_data[64 + 1] = { 0 };
for (u32 i = 0, j = 0; i < 32; i++, j += 2)
{
  u8_to_hex (encrypted_data[i], hex_encrypted_data + j);
}

const int line_len = snprintf (line_buf, line_size, "%08x%08x%08x%08x:10000:%s",
byte_swap_32(salt->salt_buf[0]),
byte_swap_32(salt->salt_buf[1]),
byte_swap_32(salt->salt_buf[2]),
byte_swap_32(salt->salt_buf[3]),
hex_encrypted_data);

return line_len;
```

Explanation

- We want to retrieve the 3 elements that were inserted in the various structures in hash_module_decode() and convert them to the same format as the input.
- The elements are converted from bytes to hexadecimal so Hashcat can write them to the console and to the potfile. The number of iterations written to the output is hardcoded for simplicity. It is important to stick to the same order and structure as the input, or else the verification of cracked hashes will not work.

Hashcat uses a potfile to store hashed passwords and their associated cracked results. This ensures that hashcat avoids unnecessary work if the hash has already been cracked.

Step 7 ➥ Add the init () method

1. Add the module_init() method. This is the last method and is always positioned at the end of the module. You must specify which methods were used in the module. All other methods must be set to default (MODULE_DEFAULT).

Example module_init()

```
void module_init (module_ctx_t *module_ctx)
{
  module_ctx->module_context_size             = MODULE_CONTEXT_SIZE_CURRENT;
  module_ctx->module_interface_version        = MODULE_INTERFACE_VERSION_CURRENT;

  module_ctx->module_attack_exec              = module_attack_exec;
  module_ctx->module_benchmark_esalt          = MODULE_DEFAULT;
  module_ctx->module_benchmark_hook_salt      = MODULE_DEFAULT;
  module_ctx->module_benchmark_mask           = MODULE_DEFAULT;
  module_ctx->module_benchmark_charset        = MODULE_DEFAULT;
  . . . . . . . . . . . . . . . .
```

We have explained the most commonly used methods and you added to them to your own module. You only need to do a few more things for this workshop.

Step 8 ➥ Gone through all of the steps? Let’s compile

1. Compile and test your module.

Your module is now ready. You can leave the kernel empty for now. It is advisable to compile hashcat at this point. This has two advantages: you can check for compilation errors and ensure that the input data is processed correctly.

To do this, print the first 8 bytes of your encrypted data in hash_module_decode():

Example print line in the module

```
printf("Printing the first 8 bytes of your data : %08X %08X \n", workshop-
>encrypted_data[0], workshop->encrypted_data[1]);
```

Compile hashcat and start an attack with a small brute-force mask. You have not yet completed the kernel so the attack will not actually start. Hashcat does print the input of the self-test hash you specified. This allows you to check that your data matches your reference implementation.

Example command-line

```
./hashcat -a 3 -m 12345 example.hash ?l?l
```

Before compilation, it is important to check that you completed all parts correctly.
Use the example file module_12345.c for this.
Are you getting compilation errors? Solve these before starting on the kernel.

4. The kernel

If the compilation was successful, that's great! You can now get started on the kernel.

Step 9 ➥ Rename the template

You will create an outside kernel prior to the workshop. We have already prepared this for you. You can use the blank kernel file 'm-template-pure.cl'.

- Add your unique attack number to the file name. This is the number you chose in step 1 of this chapter.
- Move the file to the OpenCL folder in hashcat.

Step 10 ➥ Split your implementation

An outside kernel usually contains a relatively slow cryptographic algorithm. To prevent the GPU from hanging for too long, attacks are split up into three stages: init(), loop() and comp().

The names indicate what happens in each of the stages:

- The algorithm is prepared in the init() stage
- The loop() contains actions that must be repeated several times
- In the compare() stage, the system checks whether the password has been cracked

When creating your own plugin, it is important to investigate what part is the best match for the loop(). This part can be accessed several times. This will also show you what parts come before and what comes after.

You can split things up as follows for the reference implementation:

init()	Create the first MD5 from the password and salt.
loop()	Run the first MD5 another 10,000 times through MD5 to generate the AES-key.
comp()	Decrypt the encrypted data using the AES-key and check if this was successful.

Step 11 ➥ Copy the structs

1. Copy the defined structs from your module to the kernel. This ensures that you can also use the structs on the GPU.

It is important that you synchronize the struct definitions between the module and the kernel, since hashcat does not synchronise the two automatically.

Example of the struct esalt and tmps

```
typedef struct workshop
{
  u32 encrypted_data[8];

} workshop_t;

typedef struct workshop_temp
{
  u32 digest_buf[4];

} workshop_temp_t;
```

Step 12 ➥ Insert your attack number in the kernel

- Insert your unique attack number for each method so hashcat knows that it belongs to the module with the same number.

Example init(), loop() and comp() from mempty_pure.cl

```
KERNEL_FQ void m<number>_init (KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))
{
}

KERNEL_FQ void m<number>_loop (KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))
{
}

KERNEL_FQ void m<number>_comp (KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))
{
}
```

Explanation

- All three methods have “KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))” added to them. This must be specified for all three so hashcat can access your specific structs.

Good to know: hashcat manages the generation and distribution of passwords. Hashcat processes an entire block of passwords before starting on the next block.

Step 13 ➥ Enter the init() method

- Make sure that the password and salt in the init() are run through MD5 and the first four bytes are replaced with 0, 1, 2, 3.

Use the example from the m12345-pure.cl kernel to see how this should be done.

Example init() m12345-pure.cl

```
KERNEL_FQ void m12345_init (KERN_ATTR_TMPS_ESALT (workshop_temp_t,
workshop_t))
{
  const u64 gid = get_global_id (0);

  if (gid >= GID_CNT) return;

  md5_ctx_t ctx;

  md5_init (&ctx);

  md5_update_global (&ctx, pws[gid].i, pws[gid].pw_len);

  md5_update_global (&ctx, salt_bufs[SALT_POS_HOST].salt_buf,
salt_bufs[SALT_POS_HOST].salt_len);

  md5_final (&ctx);

  char *replace=(char*)&ctx.h[0];
  replace[0] = 0;
  replace[1] = 1;
  replace[2] = 2;
  replace[3] = 3;

  tmps[gid].digest_buf[0] = ctx.h[0];
  tmps[gid].digest_buf[1] = ctx.h[1];
  tmps[gid].digest_buf[2] = ctx.h[2];
  tmps[gid].digest_buf[3] = ctx.h[3];
}
```

Explanation

- Every kernel method starts with an if-statement. This serves to verify whether the gid (global id) is below a specific value. In hashcat, the gid refers to a specific password. You write the kernel as if it processes a single password.
- This kernel uses the MD5 implementation already available in hashcat.
- The code pws[gid] refers to a specific password: pws[gid].i is the password and pws[gid].pw_len is the password length.
- In replace, you can see that the first four characters of the MD5 are overwritten.
- The result is inserted in our tmp_t struct by using the gid. This is how hashcat knows which MD5 hash goes with what password.

Very convenient! Hashcat contains a comprehensive cryptographic library that you can use.

Step 14 ➥ Adjust the loop() method

- Insert the code to run the hash through the MD5 engine multiple times. For the workshop, we decided to trigger this loop 10,000 times.

Example of the loop()

```
KERNEL_FQ void m12345_loop (KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))
{
  const u64 gid = get_global_id (0);

  if (gid >= GID_CNT) return;

  u32 digest[16] = { 0 };

  digest[0] = tmps[gid].digest_buf[0];
  digest[1] = tmps[gid].digest_buf[1];
  digest[2] = tmps[gid].digest_buf[2];
  digest[3] = tmps[gid].digest_buf[3];

  for (u32 index = 0; index < LOOP_CNT; index++)
  {
    md5_ctx_t ctx;

    md5_init (&ctx);

    md5_update (&ctx, digest, 16);

    md5_final (&ctx);

    digest[0] = ctx.h[0];
    digest[1] = ctx.h[1];
    digest[2] = ctx.h[2];
    digest[3] = ctx.h[3];
  }

  tmps[gid].digest_buf[0] = digest[0];
  tmps[gid].digest_buf[1] = digest[1];
  tmps[gid].digest_buf[2] = digest[2];
  tmps[gid].digest_buf[3] = digest[3];
}
```

Explanation

- Before hashcat starts calculating, it first consults the tmp_t struct to see what is stored in there.
- At the end of the loop, the new MD5 is put back into the tmp_t struct for the next cycle. Hashcat itself triggers the loop until this has been done 10,000 times in all.

Step 15 ➥ Fill the comp()

- Enter the code to decrypt the data with AES-128-CBC and verify it for zeroes.

A lot goes on in the comp(). We split up the code and comments into three parts for this explanation.

Part 1: Copying the AES tables

```
KERNEL_FQ void m12345_comp (KERN_ATTR_TMPS_ESALT (workshop_temp_t, workshop_t))
{

  const u64 lid = get_local_id (0);
  const u64 gid = get_global_id (0);
  const u64 lsz = get_local_size (0);

  /**
   * Copy the shared aes-tables
   */
  #ifdef REAL_SHM

  LOCAL_VK u32 s_td0[256];
  LOCAL_VK u32 s_td1[256];
  LOCAL_VK u32 s_td2[256];
  LOCAL_VK u32 s_td3[256];
  LOCAL_VK u32 s_td4[256];

  LOCAL_VK u32 s_te0[256];
  LOCAL_VK u32 s_te1[256];
  LOCAL_VK u32 s_te2[256];
  LOCAL_VK u32 s_te3[256];
  LOCAL_VK u32 s_te4[256];

  for (u32 i = lid; i < 256; i += lsz)
  {
    s_td0[i] = td0[i];
    s_td1[i] = td1[i];
    s_td2[i] = td2[i];
    s_td3[i] = td3[i];
    s_td4[i] = td4[i];
    s_te0[i] = te0[i];
    s_te1[i] = te1[i];
    s_te2[i] = te2[i];
    s_te3[i] = te3[i];
    s_te4[i] = te4[i];
  }

  SYNC_THREADS ();

  #else

  CONSTANT_AS u32a *s_td0 = td0;
  CONSTANT_AS u32a *s_td1 = td1;
  CONSTANT_AS u32a *s_td2 = td2;
  CONSTANT_AS u32a *s_td3 = td3;
  CONSTANT_AS u32a *s_td4 = td4;

  CONSTANT_AS u32a *s_te0 = te0;
  CONSTANT_AS u32a *s_te1 = te1;
  CONSTANT_AS u32a *s_te2 = te2;
  CONSTANT_AS u32a *s_te3 = te3;
  CONSTANT_AS u32a *s_te4 = te4;

  #endif

  if (gid >= GID_CNT) return;
```

Explanation

- The section on init() explained why we usually put everything below the if-statement. In some cases it is different and this is one of those cases.
- AES decryption works with set tables that you must copy onto the GPU. A group of threads on the GPU can collaborate to prepare these tables. This requires *all* threads to collaborate, not just those allocated to a password, so the gid is not checked until after the copying operation.

Part 2: Decryption

```
  // aeskey = first 16 bytes of decrypted_crypt_key
  u32 aes_key[4];
  aes_key[0] = tmps[gid].digest_buf[0];
  aes_key[1] = tmps[gid].digest_buf[1];
  aes_key[2] = tmps[gid].digest_buf[2];
  aes_key[3] = tmps[gid].digest_buf[3];
  // iv = only zeroes
  u32 iv[4] = { 0 };

  u32 local_encrypted_data[4];
  local_encrypted_data[0] = esalt_bufs[DIGESTS_OFFSET_HOST].encrypted_data[0];
  local_encrypted_data[1] = esalt_bufs[DIGESTS_OFFSET_HOST].encrypted_data[1];
  local_encrypted_data[2] = esalt_bufs[DIGESTS_OFFSET_HOST].encrypted_data[2];
  local_encrypted_data[3] = esalt_bufs[DIGESTS_OFFSET_HOST].encrypted_data[3];

  u32 decrypted_data[8] = {0}; // 32 bytes
  u32 ks[44];

  aes128_set_decrypt_key (ks, aes_key, s_te0, s_te1, s_te2, s_te3, s_td0, s_td1, s_td2,
s_td3);

  aes128_decrypt (ks, local_encrypted_data, decrypted_data, s_td0, s_td1, s_td2, s_td3,
s_td4);
```

Explanation

- Hashcat retrieves the data required for decryption. In this case, the generated aes_key from the tmp_t struct and the decrypted data from our own struct: the esalt. If multiple hashes are attacked simultaneously, the DIGESTS_OFFSET_HOST is used to refer to the correct esalt.
- This also contains the reference to aes-128-cbc that handles the decryption and uses the tables we copied. The result is placed in decrypted_data[4].

Part 3: Checking the result

```
  if ((decrypted_data[0] == 0) && (decrypted_data[1] == 0) && (decrypted_data[2] == 0)
&& (decrypted_data[3] == 0))
  {
    if (hc_atomic_inc (&hashes_shown[DIGESTS_OFFSET_HOST]) == 0)
    {
      mark_hash (plains_buf, d_return_buf, SALT_POS_HOST, DIGESTS_CNT, 0,
DIGESTS_OFFSET_HOST + 0, gid, 0, 0, 0);
    }
  }
```

Explanation

- In this last step, hashcat checks whether the decryption process proceeded correctly *and* whether we found the correct password. It does so by checking whether the first 16 bytes (first 4 elements of decrypted_data) equal 0.

- If the first 16 bytes are checked, the probability of a false positive outcome is very small, so we do not have to check all 32 bytes of the decrypted data.
- We use the mark_hash() method to tell hashcat that the hash was found and what password is associated with it.

Ready? Let’s attack!

Did you execute and verify all of the steps? Then you can now compile hashcat and run your own attack. We expect that it will not go smoothly the first time and you will get compilation errors. That’s entirely normal. The compiler usually tells you clearly what went wrong. So read your error messages and then update your code.

Do not hesitate to look at the examples and use them if you do not know what to do. The idea is for you to end up with a working plugin, not to get bogged down by the syntax of OpenCL or C.

Tip: Use print lines

It makes sense in the course of development to check, for each step, whether the right outcome is returned *and* whether the byte sequence is correct for the next step. You can use print lines to achieve this. If you do not get the desired outcome, you will know exactly in which step things go awry and where you need to edit your code.

Example print line in the console

```
printf("Printing your hash here : %08X %08X \n", ctx.h[0], ctx.h[1]);
```

Autotuning generates a lot of noise on the console if you use print lines. To prevent this, you must make sure the kernel cannot be successful. Autotuning is not done if a self-test fails. This is useful to check that your input/output and byte sequence are correct without being unduly bothered by a lot of noise.

Conclusion!

You have now worked through all of the steps needed to add a plugin to hashcat. We want to add two last bits of advice.

1. Always try to get your plugin to work correctly before doing anything else. So do not get too hung up on the performance at first, getting the right answer is the main thing.

2. Is your plugin working? Now you can focus on performance. Cracking passwords is a high-intensity computing operation. A faster plugin equals shorter cracking times, which is crucial for your attack’s deployability.

You can improve its performance through the cryptographic algorithm, GPU memory usage and low-level tweaks. If you do not know where to start, have a look at a similar plugin in hashcat. This can give you a reliable indication of what can be achieved in terms of performance.

Have any questions or ideas? Or would you like us to assist you? Feel free to let us know!

Pelle Kuiters & Ewald Snel

https://github.com/fse-a

Vital hashcat terminology

Hash mode	‘Hash mode’ means the method or algorithm that is used to convert passwords to hash values and vice versa. You need to know a password’s hash mode to target your attack. Different applications and systems use different hash algorithms to store and validate passwords.
Kernel	OpenCL code that is executed on the GPU. Mostly contains the cryptographic operations that are required to generate the correct answer.
Module	Code in C containing the configuration of your plugin, such as the description, length and a self-test hash.
Plugin	This is the term we use to describe your attack. A plugin always contains at least a kernel and a module. Sometimes there are also additional files as extra tooling.
Temporary structure (tmp_t)	This is the struct we use to share data between the various kernel methods. It is used to save and share intermediate results. This struct is intended exclusively for use on the GPU.
Embedded salt structure (workshop_t)	This is the struct we use to copy data from the CPU to the GPU and use it on the GPU. A piece of encrypted data or a key you need in a specific stage to verify something, for instance.