Silence Will Fall (Or How It Can Take 2 Years to Get Your Vuln Registered)

BI.ZONE 18 min read · Jan 18, 2021

By Innokentii Sennovskii

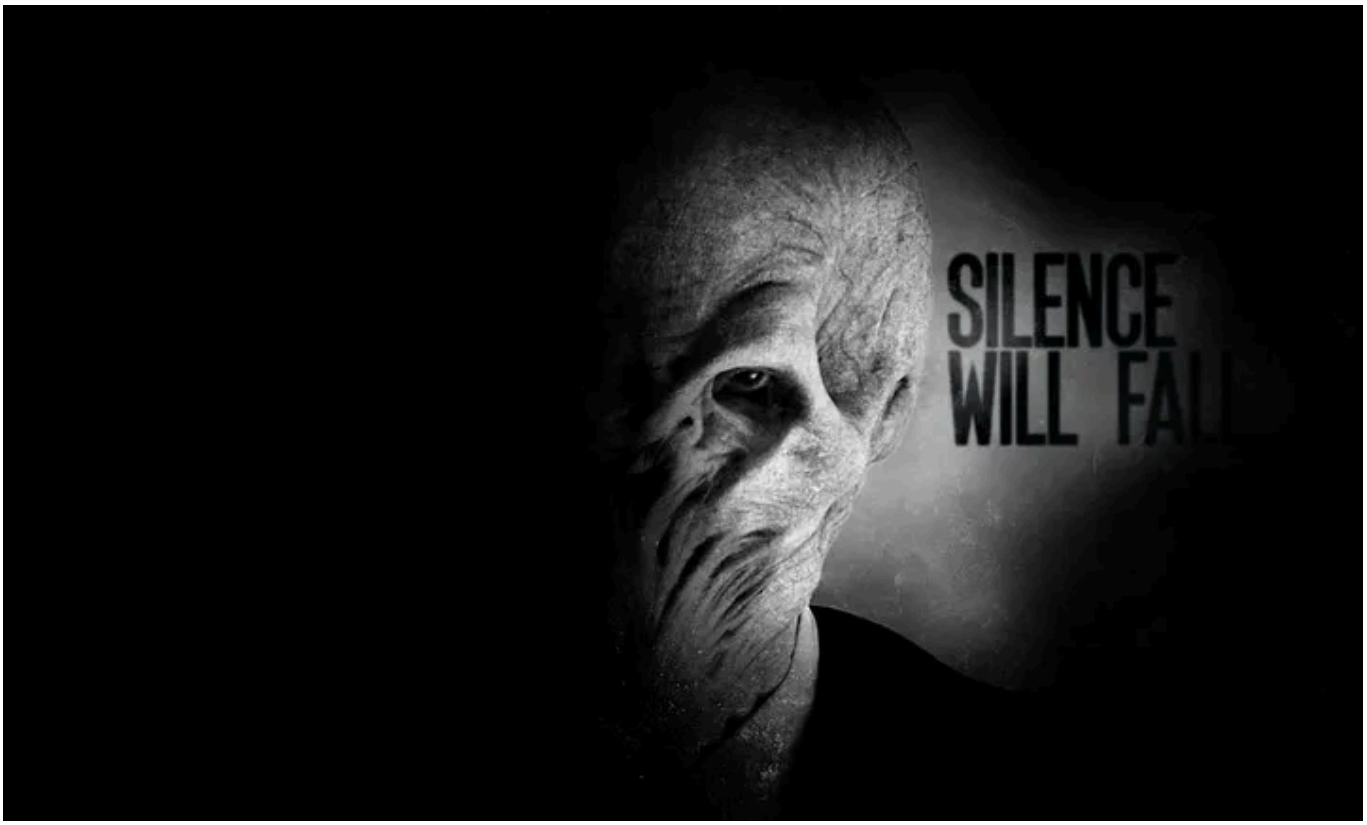

Photo credit: https://www.deviantart.com/heroeswho

How long can it take to register a vulnerability? We are quite used to bug bounties, where most of the time you just have to wait for the company in question to triage your report, and that's it, now you have a CVE. However, if the company doesn't have an established framework for submitting security issues, then all your attempts can be met with a lack of understanding and sometimes even pushback. This is a story of how I discovered two similar vulnerabilities in cryptographic libraries for embedded devices and how it took two years to get them registered. If you don't care about the story and just want to see PoC description, check out the chapter below.

Intro

In 2018, we at BI.ZONE were preparing CTFZone Finals (held offline) that were meant to coincide with the OFFZONE conference. We really wanted to spice up the competition, so we decided to host the PWN challenge on actual hardware. Since this was an Attack-Defense CTF (that's where teams are given identical services to host and protect, and all the teams try to steal flags from each other), we could give each team a custom circuit board with the CTFZone logo. The circuit board had dual purpose: a somewhat unusual task and a piece of memorabilia. At the start of the project, when the deadlines were still far ahead, we imagined a design, where the board would host two chips: one containing the core logic and one just for authentication. The attackers would be able to upload their malicious code to the core chip's memory by leveraging the embedded vulnerabilities, but it was impossible to break into the authentication chip. This way we could prevent the attacking team from simply cloning the defending team's device once and being able to emulate it later. Instead they would have to interact with the authentication chip on the target device in order to get the flag for each round. Such mechanisms require cryptographic solutions. Thankfully (or so I thought), ST (we were going to use STM chips for both microcontrollers) provided a precompiled cryptographic library for its devices that is called

STM32 cryptographic firmware library software expansion for STM32Cube (X-CUBE-CRYPTOLIB).

So I downloaded the documentation and started scrolling through the provided algorithms. I came across the descriptions of various Elliptic Curve Cryptography Primitives and was nearing SHA-1, which I wanted to try out to check if the API was working at all, but then something utterly disturbing caught my eye when skimming the RSA chapter. The firmware library contained four exported functions that you can see in the following table.

Function name	Description
RSA_PKCS1v15_Sign	PKCS#1v1.5 RSA Signature Generation Function
RSA_PKCS1v15_Verify	PKCS#1v1.5 RSA Signature Verification Function
RSA_PKCS1v15_Encrypt	PKCS#1v1.5 RSA Encryption Function
RSA_PKCS1v15_Decrypt	PKCS#1v1.5 RSA Decryption Function

This was in 2018, and at the time I thought that PKCS#1v1.5 had already died out, since the specification contains a vulnerability. The discovery of this led to the realization that faulty legacy implementations can still be found in the wild. Let's look into the reasons why PKCS#1v1.5 must not be used, and then get back to the disclosure chronology.

PKCS#1v1.5

PKCS#1 is the RSA Cryptography Standard. It defines the primitives and schemes for using the RSA algorithm for public-key cryptography. Its current version is 2.2, but the version used in the X-CUBE-CRYPTOLIB is 1.5. This

version of the specification is infamous for introducing a critical vulnerability into the encryption / decryption process. However, to understand it in full we must first cover the inner workings of RSA and what is inside PKCS#1.

Basic RSA

The RSA primitive relies on the hard problem of factorizing a product of two large prime numbers. Imagine Bob wants to send Alice a message that only she would be able to read. They decide on the following algorithm (all the formulas will follow the list, since medium can't handle Latex):

1. Alice randomly picks two prime numbers (p and q) of predefined length (for example, 1024 bits).
2. Alice computes their product N (the modulus).
3. Alice takes the predefined public exponent e. The most common value is *65537*, because it has just 2 bits set. In the past *3* and *17* have also been used, but they are not guaranteed to wrap the modulus during exponentiation.
4. Alice computes private exponent d. Now (d, N) is Alice's private key and (e, N) is Alice's public key.
5. Alice sends her public key to Bob.
6. Bob converts the message M which he wants to send to Alice into a number m less than N. He exponentiates the number to e modulo N and sends the result to Alice.
7. Alice takes the result, exponentiates it to d modulo N and gets the initial m as a result, which she then converts back into M
   ! On a side note, since Medium does not support Latex, some exponents

are expressed in brackets using the ^ symbol.

$N=p*q$, where p, q are prime

$e*d = 1(mod\ (p-1)*(q-1))$

$\forall m \neq 0: m\ ^(e*d)\ (mod\ N)=m^(1+k*(p-1)*(q-1))\ (mod\ N)=m\ (mod\ N)$, where $k \in N$

RSA teething problems

RSA has several problems. For example, you can't just pick any two prime numbers to compute the modulus, they need to fulfill certain requirements. Otherwise, the primitive becomes unsafe. And even if you've picked the perfect p and q, you'd still be left with some issues. We'll use the following notation from now on: m stands for plaintext (message) and c stands for ciphertext.

First, what if instead of using *65537* as the public exponent, you pick a smaller one such as *3*? If p and q are both 1024 bits, then the modulus is *2048* bits or *256* bytes. If the message Bob wants to send to Alice is less or equal to

$\lfloor 256/3 \rfloor = 85$

then the result of the exponentiation operations doesn't overflow the modulus. In which case decrypting the ciphertext is easy, you just need to take its third root.

Second, if Bob is repeating certain messages, then they encrypt to the same ciphertext and as a result an adversary performing a Man-in-the-Middle attack can extract partial information about the messages: the fact that at particular points in time Bob sent the same messages.

Specification

To address these issues, RSA (as in the company) created a specification that became PKCS#1 v1.5 (Public Key Cryptography Standards). The specification introduces formatting for encrypted (or signed) blocks (effectively, a special padding). This padding is used before the whole block is encrypted. Let's say the length of modulus *N* in octets (bytes) is equal to *k*. The data we want to encrypt/sign is *D*. Then the encryption block is:

```
EB = 00 || BT || PS || 00 || D
```

where *BT* is *00* or *01* for private-key operations (signing) and *02* is for public-key operations (encryption). The encryption block is translated to integer using big-endian conversion (the first octet of EB is the most significant one) *PS* consists of *k-3-len(D)* octets. Its contents depend on *BT*:

1. *BT=00*, all octets in *PS* are set to *00*
2. *BT=01*, all octets in *PS* are set to *FF*
3. *BT=02*, all octets in *PS* are pseudorandomly generated and not equal to *00*

It is recommended to use *BT=01* for signatures and *BT=02* for encryption, since

1. With *BT=01*, *D* can be unpadded regardless of its contents, with *BT=00* the first byte of *D*, which is equal to *00*, will also be unpadded, thereby creating a problem.
2. Both *BT=01* and *BT=02* create large integers for encryption, so all attacks requiring a small (short) *D* don't work

You can also notice, that the first octet of *EB* is *00*. This value was chosen so that the integer resulting from *EB* is always less than the modulus *N*.

At first glance, this scheme seems pretty reliable and it obviously solves the problems mentioned earlier.

Bleichenbacher's Padding Oracle Attack

Unfortunately, not all is well and good with PKCS#1v1.5, or I would not be sitting here writing this article. The following attack is 22 years old.

You can find the original paper here: Chosen Ciphertext Attacks Against Protocls Based on the RSA Encryption Standard PKCS#1.

Now, let's think back to Alice and imagine that she has already created the key pair and sent Bob the public key, but now Bob and Alice are using PKCS#1v1.5 padding in their encrypted messages. What happens when you send Alice a ciphertext?

1. Alice decrypts the ciphertext with the RSA primitive.
2. Alice unpads the message.
3. Alice parses the message and performs actions based on the message.

Various programs and systems (Alice in this analogy) tend to signal to the sender (Bob), that something has gone wrong if there is a problem with receiving commands. The issue is that these signals quite often show exactly where the problem has arisen: during the second stage (padding is incorrect) or the third stage (something wrong with the command itself). Usually, discerning stages two and three is only possible with a private key, so when

Alice tells us whether unpadding failed or succeeded, she actually leaks information about the plaintext.

As a thought experiment, what if we send a random integer *c* for decryption instead of a legitimate message? What is the probability that no error will surface during unpadding? The first 2 octets of *EB* should be *00* || *02*, the other octets should have at least one *00*. If *k* is 256, then $P=1/256^2 \cdot (1 - (255/256^{254}))\approx 1/104031$. The probability seems way too small, but it is actually high enough to be leveraged in an attack. The attack uses the fact that RSA is homomorphic with regards to multiplication:

$$\forall x: (x*m)^e \pmod N = x^e * m^e \pmod N$$

Here is how the attack unfolds. We choose the *c*, for which we want to find

$$m = c^d \bmod N$$

For the sake of convenience, we'll use the following constants in further explanation

B=2^(8(k-2)), k=||N||.

We say that ciphertext is PKCS-conforming if it decrypts to a PKCS conforming plaintext. Basically, we choose integers *s*, compute

$$c' = cs^e \bmod N$$

and send them to the oracle (Alice). If *c'* passes the error check, then

$$2B \leq ms \bmod N < 3B.$$

By collecting enough different s we can derive m, this typically requires around 2^{20} ciphertexts, but this number varies widely. The attack can be divided into three phases:

1. Blinding the ciphertext, creating c_o, that corresponds to unknown m_o
2. Finding small s_i, such that $m_o s_i \bmod N$ is PKCS-conforming. For every such s_i the attacker computes intervals that must contain m_o using previously known information
3. Starts when only one interval remains. The attacker has sufficient information about m_o to choose s_i such that $m_o s_i \bmod N$ is more likely to be PKCS conforming than a randomly chosen message. The size of s_i is increased gradually, narrowing the possible range of m_o until only one possible value remains.

(Actually, when the interval is small enough, it is more efficient to just bruteforce the values locally than continue submitting queries to the oracle).

So, the model of the attack is:

1. Intercept some message from Bob to Alice.
2. Use Alice as a padding oracle to decrypt the message.

Back to disclosure

Since I found this vulnerability in one software library for microcontrollers, I decided to check if there are any other libraries that provide this specification as the default method for RSA encryption/decryption. Turns out, ST is not the only company. Microchip also provides a library that contains cryptographic primitives and lo and behold — the only encryption

scheme for RSA is once again PKCS#1v1.5. Here is an excerpt from the RSA decryption function:

Here, the order of bytes in memory is inversed, so we actually check that the plaintext ends with *0200* instead of starting with *0002*. Anyway, we can see that the function returns a different status when the padding is incorrect and this is what makes it dangerous.

So it was time to alert the manufacturers. I submitted the descriptions to ST and Microchip in September of 2018 and started the waiting game.

In the beginning ST was quite responsive, but there was some miscommunication. Part of it was on me, since I initally wrote that the fault was in STM32-CRYP-LIB (another cryptographic library provided by ST) instead of X-CUBE-CRYPTOLIB (and the problem is that STM32-CRYP-LIB doesn't provide encryption/decryption primitives for RSA, only signing/verification). There was also the regular pushback associated with Bleichenbacher's padding oracle in libraries, namely, that it doesn't necessarily mean that the end product will contain a vulnerability, but they agreed that it is hard to mitigate this problem once you've chosen to use PKCS#1v1.5. In October I was told that they have started work on implementing PKCS#1v2.2. I emphasised the need for a CVE to alert users, but their next responce came only in December:

- They didn't want to register a CVE, because they just followed the specification and the library itself didn't return any padding errors (this is just partially true)
- They were developing PKCS#1v2.2 (a newer specification with safer padding) and were going to publish it with a new version of X-CUBE-CRYPTOLIB in the spring.

I tried to change their mind on registering a CVE, but in the end couldn't.

I submitted the issue to Microchip in September. The Development team only got back to me with a "solution" to my problem in March of 2019. They advised me to use Microchip Harmony (their Embedded Software Development Framework) instead of Microchip Libraries for Applications package. I must admit that it is indeed a valid solution, since Harmony

contains more modern specifications that can be used instead of the vulnerable one. However, there still was a need to alert users about the pitfalls of using RSA in Microchip Libraries for Applications and it took almost 6 months for them to respond to my initial request, so I decided to take a different route.

MITRE requires you to try every possible method for registering a CVE before applying directly to them. By March of 2019, I tried to contact the vendors, but that didn't help with the registration. After that the vulnerabilities is supposed to be submitted to one of the CVE Numbering Authorities, but none except MITRE were in fact relevant. However, HackerOne is actually a CNA and they have a program called "The Internet". To qualify for it you have to either find a vulnerablity in some widely-used open-source project, or the vulnerability should affect several vendors (which is my case). However, now, there was a need for a Proof-of-Concept (PoC). So, I set out to create just that.

Proof of Concepts

General information

Since these libraries are aimed at embedded devices, using them on a regular PC with amd64 architecture comes with certain limits. The worst of them is the speed of computation.

The MLA is distributed as source code, but the actual bigint computations are implemented in PIC architecture assembly. I had to rewrite them to amd64 architecture, but suffered a slowdown, since the procedures in C code relied of 16-bit computations in assembly routines. This makes RSA quite

slow, even though it is executed “natively”. It can take up to several days to complete the attack. On a device it would take significantly less.

X-CUBE-CRYPTOLIB is distributed in compiled format. To use it on a PC, I had to emulate the firmware with QEMU. Obviously, it is a significant obstacle for time. What makes it even worse is that the only available interface to transport the data is emulated UART, which is painfully slow (it takes 2 minutes to transmit 256 bytes). To deal with the interface problem, I’ve implemented a debug hook (so QEMU is running with debugging) that transfers data to and from memory. This allows us to avoid at least the transmission emulation bottleneck.

Still in these conditions the attack is quite slow. So I implemented two versions of the attack for both platforms to save the tester’s time:

- The full attack (will take a long time on MLA, an absurdly long time on ST)
- The attack with precalculated successful steps

The second variant is basically running the algorithm beforehand with a python server which performs the same checks as the vulnerable library. It saves all the steps which produce the correct decryption outcome (these steps allow the attacker to narrow the plaintext space). The attacker only tries these with the vulnerable implementation later (thereby skipping most of painful bruteforce).

```
MEMORY[0x40023008] = 1;
v22 = v21 | (4 * v21) | (16 * (v21 | (4 * v21)));
v23 = v22 | (v22 << 8) | ((v22 | (v22 << 8)) << 16);
if ( MEMORY[0x40023000] == -1 )
{
  v24 = 0;
  if ( MEMORY[0x40023000] )
  {
    while ( v24 < *(_DWORD *)(a1 + 4) )
    {
      text_592();
      v8 += ((v24 + v10 - 3) >> 25) & 1;
      ++v24;
    }
  }
  else
  {
    while ( v24 < *(_DWORD *)(a1 + 4) )
    {
      text_592();
      v8 += ((v10 - v24++ + 2) >> 31) & 1;
    }
  }
}
else
{
  for ( i = 0; i < *(_DWORD *)(a1 + 4); ++i )
  {
    text_592();
    v26 = 5 * i;
    v8 += ((v10 - v26) >> 27) & 1;
  }
}
```

It's also worth noting that the X-CUBE-CRYPTOLIB library is protected against being used on non-ST devices (the check is performed by writing and reading bits from specific memory registers). You can see them in the picture on the left (**MEMORY** accesses). QEMU fails this check, but fortunately instead of refusing to decrypt/encrypt data the routines just mask data during encryption and give back more data (including some of the random padding) during decryption. The mask can be rolled back as long as the plaintext contains only ASCII-7 characters, since all of them are less than 128. So, the unmasking function also had to be implemented to use the library with QEMU.

There was only one change made to the MLA library: implementing bigint arithmetic in intel assembly instead of PIC.

There were no changes made to the ST library.

What follows are instructions on how to reproduce the issues (on Linux).

Preliminary steps

MLA library is readily available here: https://www.microchip.com/mplab/microchip-libraries-for-applications, but you have to request X-CUBE-CRYPTOLIB in advance here (you'll have to register): https://www.st.com/en/embedded-software/x-cube-cryptolib.html. Or if you've already registered and are logged in: https://my.st.com/content/my_st_com/en/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-cryptolib.html.

To use RSA, we first need to generate a private key. The private key will be used to create the speedup trace and it will be used by the vulnerable servers.

Unpack the archive "*Bleichenbacher_on_microcontrollers*.tar.gz" (available here). Go to subfolder "*Preparation*".

List of files in *"Preparation"*:

- requirements.txt (file with python requirements for *"create_traces_and_imports.py"* and python attack clients)
- create_traces_and_imports.py (python script which parses a pem file, creates headers and imports for vulnerable servers and clients, also encrypts one message, which is to be decrypted by the attacking clients and creates a trace to show the attack without taking too much time)

- initialize.sh (runs openssl to create a 2048-bit RSA key, then runs *“create_traces_and_imports.py”*)

You need to install openssl, python3, pip, python3-gmpy2 and pycryptodome. If on ubuntu, run:

```
sudo apt install openssl python3 python3-pip python3-gmpy2 && python3 -m pip install -r requirements.txt
```

Now, you can run the initialization file which will create the keys, the trace and the headers for the future use. “initialize.sh” first creates an RSA key “private.pem” and then runs the “create_traces_and_imports.py”. The files contain comments, so you can look inside to see what’s happening.

Run:

```
./initialize.sh
```

Files and folders created:
microchip_files
- attacker_params.py
- key_params.h

speedup
- info.txt
- trace.txt

The “info.txt” file contains information on how long it would take for the client to complete each variant of attack (based on my hardware, yours can be better).

Microchip PoC

Install Microchip Libraries for Applications (MLA). The latest version at the time of writing is “v2018_11_26”. It has to be installed in a non-virtualized environment. For some reason they’ve implemented this check in the installer.

Go to the directory where you’ve extracted all files from “Bleichenbacher_on_microcontrollers.tar.gz” (available here). Go to “Microchip/vulnerable_server”.

List of files in the directory:

- bigint_helper_16bit_intel.S (rewritten *“bigint_helper_16bit.S”* from the bigint library inside MLA. It was written for PIC architecture, making the library unusable on a PC. I rewrote all functions in intel (x86_64) assembly, to use the same bigint library that crypto_sw in MLA uses)
- bleichenbacher_oracle_mla.c (the main file containing high-level server functions)
- crypto_support_params.h (enums and function definitions for cryptographic functions)
- crypto_sw.patch (a patchfile that comments out one line in crypto_sw library to stop type collision when building on a PC)
- crypto.c (contains all functions which initialize and deinitialize MLA’s RSA implementation and functions that wrap around encryption and

decryption routines)

- do_patch.sh (a simple bash script to apply the patch)
- makefile
- oracle_params.h (some server parameters)
- support.c (decrypted message checking function)
- support.h (check_message function and return values' definitions and seed length definition)

Copy the folders "*bigint*" and "*crypto_sw*" from "~/microchip/mla/v2018_11_26/framework" to "*microchip_build*". Copy the file "*key_params.h*" from "*Preparation/microchip_files*" in the initialization phase to the *"Microchip/vulnerable_server"*. Run *do_patch.sh*. It will change one line in the crypto_sw library (there is a type definition collision because we are using system's stdint). And finally you can run make.

```
cd Microchip/vulnerable_server
cp -r ~/microchip/mla/v2018_11_26/framework/bigint .
cp -r ~/microchip/mla/v2018_11_26/framework/crypto_sw .
cp ../../Preparation/microchip_files/key_params.h .
./do_patch.sh
make
```

The file "*bleichenbacher_oracle_mla*" is the program containing the vulnerable oracle server using microchip's library.

Now, open another terminal (let's call it "AttackerTerminal" and the previously used one "VictimTerminal"). Go to *"Attacker"* in this newly created terminal and copy *"Preparation/attacker/attacker_params.py"* to *"Attacker"*. You

can also copy the tracefile if you want to speedup the attack. Look in the *"Preparation/speedup/info.txt"* to see how much time you'd have to spend on each version of the attack (without speeding up/skipping the initial bruteforce/just checking the trace) with the current key and encrypted message.

```
#AttackerTerminal
cp ../Preparation/attacker/attacker_params.py .
cp ../Preparation/speedup/trace.txt .
```

Now, open the victim terminal and run *bleichenbacher_oracle_mla*. It will open port 13337 and listen for one incoming connection. This means that if you want to rerun the attack, you have to stop the attacker and rerun the server, then run the attacker script again.

```
#VictimTerminal
./bleichenbacher_oracle_mla
```

You have 3 options depending on how you want to run the attack:

1. Run the full attack (you will have to wait a long time, however)

```
#AttackerTerminal
python3 attacker.py
```

2. Skip the first part of the attack (finding the first s. This is the longest part of the whole attack)

```
#AttackerTerminal
python3 -t trace.txt -s
```

3. Run only the successful queries (taken from the trace). The quickest version of the attack.

```
#AttackerTerminal
python3 -t trace.txt
```

After the attack completes, the attacking script will print the decrypted message, it will login on the vulnerable server and ask it to give the flag, printing the final answer from the server.

ST PoC

Download and unzip qemu_stm32.

```
wget https://github.com/beckus/qemu_stm32/archive/stm32_v0.1.2.zip
unzip stm32_v0.1.2.zip
cd qemu_stm32-stm32_v0.1.2
```

Install the necessary packages:

```
sudo apt install arm-none-eabi-gcc-arm-non-eabi-newlib gcc
```

Configure and build qemu:

```
./configure --extra-cflags="-w" --enable-debug --target-list="arm-
softmmu"
make
```

If there is a problem with linking which stems from undefined references to symbols “minor”, “major”, “makedev”, then you need to add the following include to the file *“qemu_stm32-stm32_v0.1.2/hw/9pfs/virtio-9p.c”*:

```
#include <sys/sysmacros.h>
```

The error is due to the changes in libc after version 2.28.

You can do this easily, like so:

```
cd qemu_stm32-stm32_v0.1.2
cp ../stm32_qemu.patch .
cp ../do_qemu_patch.sh .
```

Now, just configure and make it again.

The qemu executable, that can execute images for STM, is located in *“qemu_stm32-stm32_v0.1.2/arm-softmmu”* and is named *“qemu-system-arm”*.

Now, let’s build the image. Go back to *“ST/vulnerable_server/binary”*. Download the stm32_p103_demos archive and unzip it.

```
wget https://github.com/beckus/stm32_p103_demos/archive/v0.3.0.zip
unzip v0.3.0.zip
```

Put the X-CUBE-CRYPTOLIB archive *“en.x-cube-cryptolib.zip”* in the *“ST/binary”* and unzip it. Copy the library files for STM32F1 to the *“stm32_p103_demos-0.3.0”*:

```
unzip en.x-cube-cryptolib.zip
cp -r
STM32CubeExpansion_Crypto_V3.1.0/Fw_Crypto/STM32F1/Middlewares/ST/STM
32_Cryptographic stm32_p103_demos-0.3.0/
```

Now, you need to apply the patch containing vulnerable server’s source code and changes to makefile and the linkage file (without it the compilation will fail during linkage).

```
cd stm32_p103_demos-0.3.0
cp ../do_demos_patch.sh .
cp ../stm32_p103_demos.patch .
./do_demos_patch.sh
```

The list of files in the folder *“demos/bleich/”*:

- main.c — the main file containing high-level functionality of the server
- stm32f10x_conf.h — configuration file
- support.c — message parsing and fixing
- support.h — definitions for the use of support.c

You need to copy the parameters of the key:

```
cp ../../../../Preparation/st_files/key_params.h demos/bleich/
```

Now, you can build the vulnerable server.

```
make bleich_ALL
```

Bear in mind that if you create a new key and run create_traces_and_imports.py you will need to perform *make clean* first to rebuild with new parameters.

For this one you'll need 3 terminals:

- Server QEMU terminal
- Server GDB terminal
- Attacker terminal

In the first terminal, go to the *"<some_prefix>/ST/vulnerable_server/binary/qemu_stm32-stm32_v0.1.2/arm-softmmu"* folder and start qemu with gdb server:

```
#Server QEMU terminal
./qemu-system-arm -M stm32-p103 -kernel ../../stm32_p103_demos-
0.3.0/demos/bleich/main.bin -gdb tcp::3333
```

In Server GDB terminal do the following:

- Go to *"<some_prefix>/ST/vulnerable_server/"*
- install gdb-multiarch if it's not installed
- run gdb-multiarch
- load file *"binary/stm32_p103_demos-0.3.0/demos/bleich/main.elf"* (this is necessary for the script to work)
- connect to qemu gdb server
- load python script *"<some_prefix>/ST/vulnerable_server/gdb_server/python-extender.py"* (gdb has to use python3)

```
#Server GDB terminal
gdb-multiarch
file binary/stm32_p103_demos-0.3.0/demos/bleich/main.elf
target remote localhost:3333
source gdb_server/python-extender.py
```

At this point, the terminal should hang. This is ok, it's waiting for a connection from the server.

The steps to attack the server are the same as in Microchip PoC. If you want to do the attack again, first you have to stop qemu, then you need to kill gdb.

```
pkill -9 gdb
```

Unfortunately, it can't exit itself because of python threads. After you've killed the server you can restart the emulator and gdb server in the same

order.

Back to disclosure... Again

So, I submitted these PoCs to HackerOne in October of 2019. However, since Bleichenbacher's Padding Oracle Attack has only Medium severity, the impact wasn't high enough for the report to be eligible for the Internet program. Sigh... Ok, the only thing that was left was to submit the vulnerabilities directly to MITRE. On the 21st of October I submitted the request and got the automatic reply. A few days after that I replied to the letter, asking if they required any more information, but there was silence. I got preoccupied with work and CTF preparations (a year has passed) and completey forgot about that letter.

In December of 2019, ST got back to me with an update: that they were to publish a new version with PKCS#1v2.2 in the beginning of 2020 and the the user manual for X-CUBE-CRYPTOLIB was going to be updated with a warning regarding the vulnerability. To this day there is no update to the library (although they did later mention that they would only provide the update for the modern STM32 families). But they did add a warning to their documentation, so at least this story had some positive outcome:

Caution: Due to PKCS # 1V1.5 specifications, there is a possibility of attack. For more details refer to PKCS # 1V1.5 specifications.

15.2 RSA library functions

Table 177. RSA algorithm functions of the STM32 crypto firmware library

Function name	Description
RSA_PKCS1v15_Sign	PKCS#1v1.5 RSA Signature Generation Function
RSA_PKCS1v15_Verify	PKCS#1v1.5 RSA Signature Verification Function
RSA_PKCS1v15_Encrypt	PKCS#1v1.5 RSA Encryption Function
RSA_PKCS1v15_Decrypt	PKCS#1v1.5 RSA Decryption Function

Later, in October of 2020, somebody reminded me about these vulnerabilities. I resubmitted the request through my work email. Once again, MITRE were silent even after I replied to the automatic response two weeks after receiving it. The situation was preposterous. At this time I looked at my first email to them and noticed, that a year has passed since I first submitted the vulnerabilities for numbering. And we all know that only one thing can get the ball rolling when you've found yourself in such a predicament: a Twitter rant. To be fair to MITRE, once I mentioned them in a tweet they quickly reserved CVEs and replied to my old emails. So, in conclusion, here are the CVEs:

1. CVE-2020–20949 (Vulnerability in ST X-CUBE-CRYPTOLIB 3.1.0, 3.1.2)
2. CVE-2020–20950 (Vulnerability in Microchip Libraries for Applications 2018–11–26)

The end.

Cve Vulnerability Rsa Cryptography

Written by BI.ZONE

208 followers · 1 following

BI.ZONE is an expert in digital risks management. We help organizations develop their businesses safely in cyberspace.