[HITB GSEC CTF 2017]
[Writeup]
[Team: PseudoRandom]

CRYPTO: HACK IN THE CARD I

Theoretical Idea

We receive a graph that represents the voltage in the card while it's decrypting some data with its private RSA key.

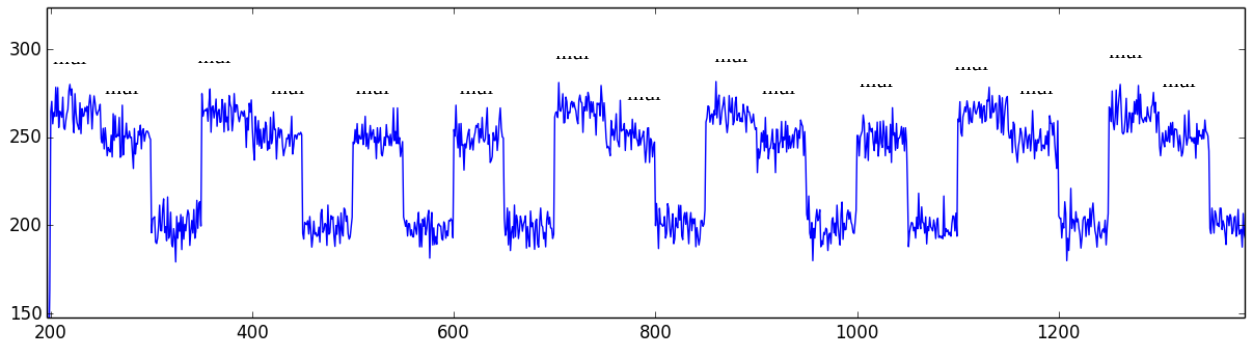
RSA decryption of a ciphered message *c* with a key *d* is done by the exponentiation of *c* by *d*: c^d

Because computers nowadays don't have some magic POW opcode that allows them to quickly exponentiate numbers, this has to be done by multiplication.
Now, in theory, we can multiply *c*, *d* times by itself. But it's horribly inefficient.

```
def pow(d, n):
    next_result = 1
    result = 1
    d_bits = bin(d)[2:]
    for b in d_bits:
        result = next_result
        if int(b) & 1:
            result = (res * m) % n
        next_result = (res * res) % n
    return result
```

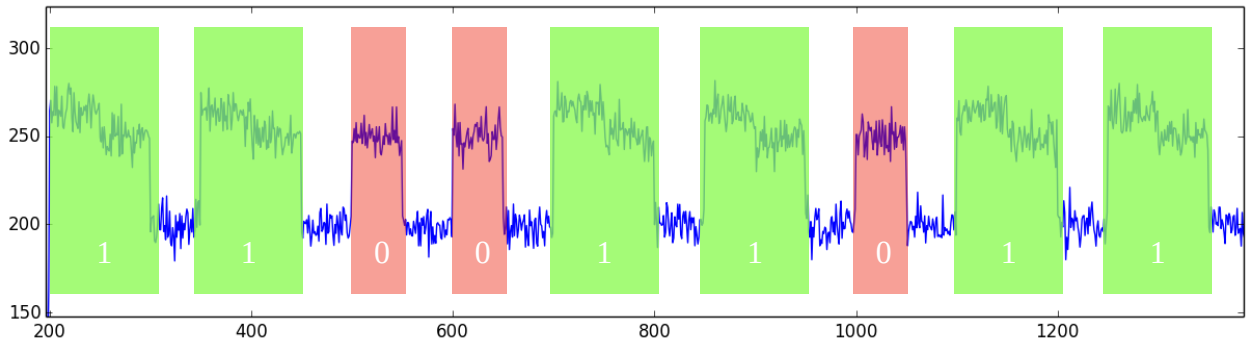
That's why exponentiation would usually be implemented this way:

As you can see, we essentially iterate the bits of the exponent, and for each bit we do one multiplication. If the bit is on, we do another multiplication.



And this is reflected in the graph:

What we see here is a bunch of "mountains", each representing either one or two multiplications.
Wide mountains represent two multiplications, meaning there was a "one" bit in the exponent, and narrow mountains represent a single multiplication, meaning we had a "zero" bit.

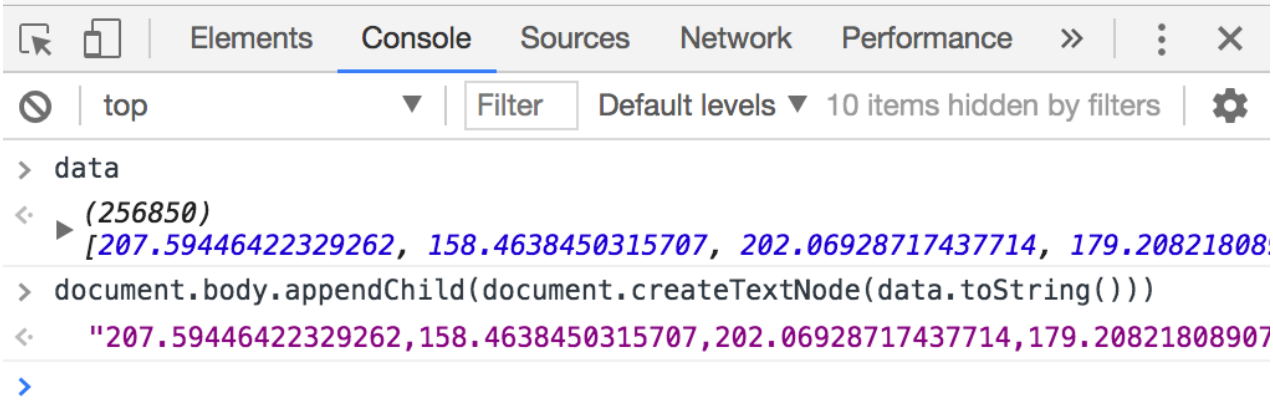


In fact, we have a total of 2048 “mountains”. Coincidence? I think not!
This is because the exponent that we derive from this graph is exactly the private key we needed, d.

Practical Implementation

0) EXPORTING THE GRAPH

First we'd like to move the graph data to a proper working environment, such as Python.
Upon inspecting the JavaScript in the supplied webpage, we can see that the graph points are stored in a variable called ***data***, which we can export using Chrome’s console:



1) DERIVING D

```
def iter_mountains(data, pivot=220, min_size=5):
    """
    Iterate the mountains' lengths
    """
    in_mountain = True
    last_t = 0
    prev_v = 0
    for t, v in enumerate(data):
        if (in_mountain and v > pivot) or (v < pivot and not
in_mountain):
            if not in_mountain and t - last_t > min_size:
                yield t - last_t
            last_t = t
            in_mountain = not in_mountain
        prev_v = v

# The start of the data is useless nonesence
data = data[199:]
```

```
bits = [1 if i > 70 else 0 for i in list(iter_mountains(data))]

def bits2num(bits):
    d = 0
    for b in bits:
        d <<= 1
        d |= b
    return d

d = bits2num(bits)
```

2) DECRYPTING THE SECRET MESSAGE

```
public_key = RSA.importKey(open('publickey.pem').read())
private_key = RSA.construct((public_key.n, public_key.e, d))

print private_key.decrypt(cipher_text.decode('hex'))
```

CRYPTO: HACK IN THE CARD II

The N was the same, but the e changed.
We denote e_1, d_1 - the e, d from before and $e2, d2$ - the new ones.
We want to find d_2 to be able to decrypt the text and get the flag.

1) CALCULATING p, q

From e_1, d_1 we can get e_1, d_1 such that $N = p * q$ the following way:
First we write $e1 * d1 = 1 \ (mod \ (p-1) * (q-1))$
so
I) $e1 * d1 = K * (p-1) * (q-1) + 1$ [for some small K]
II) $p * q = N$
Fixing K we get equations for p, q which can be easily solved (quadratic equations)
Going through several K's got us the correct factors.

2) CALCULATING d_2

From p, q we can get d_2 by calculating $e_2^{-1} \ mod(\ (p-1) * (q-1))$.

CRYPTO: HASHINATOR

A nice vulnerability is in the following lines:

```
<CODE>
rand = struct.unpack("I", os.urandom(4))[0]
lines = 14344391 # size of rockyou
line = rand % lines
password = ""
f = open('rockyou.txt', 'rb')
for i in range(line):  # !! WHAT IF line == 0?
    password = f.readline()
</CODE>
```

There is a (relatively) high chance of getting line = 0 and then password stays "".
So bruteforcing with sending '\n' each time got us the flag quickly.

MOBILE: PRIME

Decompiling the apk and looking in MainActivity.java we see that the flag that should be printed looks as follows:

```
"HITB{" + MainActivity.this.CalcNumber(MainActivity.f14N) + "}"
```

Where:

```
private static long f14N = ((long) Math.pow(10.0d, 16.0d))
```

So we should see what's the result of `CalcNumber` (10 ** 16).

The function `CalcNumber` is as follows:

```
    private long CalcNumber(long n) {

        long number = 0;

        for (long i = 1; i <= n; i++) {

            if (isOk(i).booleanValue()) {

                number++;

            }

        }

        return number;

    }
```

So it calculates the amount of number less or equals to n (in our case 10 ** 16) for which `isOk` returns true.

We check `isOk`:

```
  private Boolean isOk(long n) {

        if (n == 1) {

            return Boolean.FALSE;

        }

        if (n == 2) {

            return Boolean.TRUE;

        }

        for (long i = 2; i * i < n; i++) {
```

```
            if (n % i == 0) {
                return Boolean.FALSE;
            }
        }
        return Boolean.TRUE;
    }
```

So from first site you might think this function checks whether `n` is prime via a square root time well-known algorithm and so we should win googling the number of primes less or equal to 10 ** 16, right?

Wrong! We got “incorrect flag” for any number close enough to that amount, so we were sure we have missed something, and so we did.

Looking closer:

```
for (long i = 2; i * i < n; i++)
```

We miss out the square root! For `n = 9` we get TRUE, because `9 % 3` is not checked.
So what does that mean? Is it accepting numbers having a square root? Well not all of them, since 16 cancels out on the check vs. 2. We can accept a number if it is prime or if it has a square root which is its only divisor, meaning the square root is prime.

So we should add the amount of squares of primes less or equal to 10 ** 16, which is exactly the number of primes less or equal to 10 ** 8. (google that amount!)

And that got us the flag, hurray!

PWN: 1000LEVELS

We found 2 vulnerabilities in the binary:

0) StackO in the recursive **level** function, for the answer we have a buffer of size 0x20 on the stack, but the user can input up to 0x400 bytes (as so is the limit for the read function)

1) Uninitialized value on the stack in the **go** function, used for level calculation - on inputting 0 for the first question of levels the value is not initialized, and the answer for the second question of levels is added to it afterwards.

Interestingly, we can see that calling the **hint** function write the libc **system** function address to the same offset on the stack which can be used as uninitialized on the stack in the **go** function, granting us the ability to write any libc address in that offset on the stack through the following flow:
Hint -> Go: How many? [0] Any More? [any offset relative to system]

Ok, so now we wanted to check out the ROP we’ll write, Using /proc/[pid]/maps we saw that ASLR all over the binary, except for the unbeatable **vsyscall** page. This static page is an optimization for 3 syscalls: **sys_gettimeofday** (0xffffffffff600000), **sys_time** (0xffffffffff600400) and **sys_getcpu** (0xffffffffff600800).

Sadly, you can jump only to the exact addresses written above (and not to the syscall/ret instructions), and that also messes with the registers rdi and rsi, so we lose all control of passed values when we jump to the libc address.

But we know that a lot of functions in libc that in some flow should call **execve('/bin/sh', ..)**, just **system** is one example of such function! So indeed we found in offset -294 of system an address which when jumped to: calls **execve('/bin/sh',..)** [with some uninitialized arguments from the stack which we can hope to be zero.]

And indeed jumping to that magic address spawns a shell, and sending '/bin/cat ./flag' sends us the flag right away.

It's worth noting, that this exploit shows that for StackO-s even a leak is not needed, with "relative" control of a single libraryaddress on the stack one can spawn a shell even when there's a "complete" ASLR.

SCRIPT INCLUDED: solve_1000levels.py

PWN: BABYSTACK

We have a $SafeSEH$ program with a stackO in a certain flow.

$SafeSEH$ is MS' way of dealing with the simple SEH exploits that overwrite the SEH handler method on the stack and smash everything else. In theory, this is a great mitigation, but the problem is that in Visual C++ (the default compiler in Visual Studio), exception handled functions have a generic handler method and an extended struct on the stack containing more information for specific exception handling, and though the fields of the struct itself are somewhat protected, the values of the scope table (pointed to by one of the fields in the extended struct) aren't validated at all.

Our exploit is to leak information from the stack and use the stackO to overwrite the scope table pointer to point to a scope table in our control.

$SafeSEH$ validates the ebp chain, validates that the next field is 'relevant' (though doesn't fully follow it to it's start - that's another mitigation called $SEHOP$), but most importantly validates that the handler method pointer is in a whitelist pointed to by the PE's headers.

I think it also zeroes the general registers before calling the handler method so as to prevent leaks or whatever. The scope table pointer isn't validated, but it is XORed with the stack cookie base (that is XORed with ebp in functions with stack cookies).

What I did first was I tried to just leak the cookie base, and overwrite the scope table pointer with an address in my stack buffer (XORed, of course), as well as write the correct pointer to the SEH handler function (which we can calculate from the given address of main).

I replicated the scope table from the module, just with the jump addresses being the address of the
- push offset .. ; "cmd"
- call ds:system

in main.

Then all that's left to do is cause an exception, which we can do simply by requesting the address 0 (access violation).

Needless to mention that the image and stack addresses are given to us and we also get a 'free' absolute read, so all the leaks are trivial.

I wrote a script to do all that and ran it against the server. It failed.

I wanted to debug the program (which turned out not to be that useful as I just treated the $SafeSEH$ as a blackbox, guessing and adding more stuff to leak and overwrite correctly), but it's not trivial to do with it's IO being a socket.

I wrote a python script using Popen and stdin=sock, stdout=sock, but it didn't work. I had the idea to use nc (nc -Lvp 1337 -e babystack.exe), which worked, and then I put a raw_input call in my client script (and disabled the socket timeout) and used WinDBG to attach to the babystack process. [Just a neat trick, wasn't too useful in this pwn.]

Anyway, eventually I got it working with all the fields and managed to run cmd commands on my own machine using the exploit.
There is one small catch - the ebp chain is validated, and since the stack is almost always in an address with the top byte as zero, I assert in my script that the top byte of ebp was zero. This assert sometimes fails, but that's fine, just run it again and it's likely to work.

Once I got it to work, I ran 'dir' on the server, and got access denied, but 'type flag' worked ☺

The challenge was quite simple and just required some knowledge (or research) on $SafeSEH$.
The vulnerability is trivial, all the leaks are trivial and the exploit is just a matter of recreating the right information and overwriting the easiest thing that gives us a jump primitive, which is the scope table.

SCRIPT INCLUDED: solve_babystack.py

PWN: BABYSHELLCODE

Note: we were late just by a few minutes to submit the solution to this one, but we did all the work and got the flag so we decided to write the solution for this one too ☺

We found the following vulnerabilities here:
0) Leak of allocated allocsc address on connection start (more like a feature in scmgr)
1) StackO in name input and printing it as string ("%s") - causing leak of cookie, stack addr and babyshellcode base.
2) Leak of scmgr address - through the weird algorithm in the shellcode guard
    -> Implemented it and bruteforced through all base addresses of scmgr (2 ** 16 options, offset allways 0x1090) and checked with leaked output to find the address.
3) Relative Free - on delete shellcode can put any offset and only validates that it is not zero.
    -> Tried to exploit it for UAF in several cool ways with stack pointers and the alike, but did not succeed.
4) StackO - in the run shellcode function, the shellcode is memcpy-d to the stack via user-inputted length.
    -> After getting all the leaks - we have the address of getshell in scmgr, so to successfully exploit the overflow we had to write the cookie correctly and fake a scope table (see above) on the allocsc space so that points to the getshell function, with the scope table xored on the stack, and the correct esp/ebp/next and so on, which are built from the leaks.
    The call to it is triggered via exception of the jump to the guarded shellcode (cfg check).

SCRIPT INCLUDED: solve_babyshellcode.py