Cracking encrypted software like it's 1988

Kuba Tyszko
K6UBA
kuba@lbl.pl

whoami

root

sudo whoami

- Hacked my grandpa's farming tractor at the age of 6 with a window hook. It was left in gear, you can imagine the damages :/
- First computer - Commodore 64, attempted writing own game and ran out of memory
- Second computer - PC, optimized so much out of it that Win95 thought I had 9MB on a 8MB system !
- Cannot confirm or deny "acquiring" a payphone during the golden age of phreaking
- ~~Cracked~~ Backed up a good amount of software for personal use. You'd be surprised how many games could be tricked using windows "subst" command into thinking they're running off a CD
- Started a non-profit local ISP that operated for 13 years with over 150 devices. Thunderstorms are no joke
- Hacked into an "Internet cafe" in my hometown, was offered a job. To this day I keep my personal email account with that company
- My Bachelor's thesis was in fact about breaking AES, but there was a catch - the keys were about 20 bits long...
- Active in retrocomputing, mostly Silicon Graphics community. Have a commercial product reviving SGI Fuel machines with broken power supply. 7oz copper PCB's are hard to solder !
- Traded on Tokyo Stock Exchange without a broker license once. Don't attempt !

Problem statement

- An encrypted application
- The application is expected to run proprietary code (aka "secret sauce") in an untrusted environment. The application decrypts the code on the fly and executes it
- The code happens to be a machine learning model written in R, so the package also ships with a R runtime
- All this is bundled as a docker container (largely irrelevant)

What's in the bundle

- `./dec` `# the binary, it contains the key needed to decrypt the model`
- `./model.enc` `# the encrypted ML model`
- `./testData.txt` `# plaintext test data (not important)`

```
root@ser:~/cr# ls -la
-rwxr--r--  1 root root     20140 Sep 26 13:55 dec
-rwxr--r--  1 root root      6488 Sep 26 13:55 model.enc
-rwxr--r--  1 root root     24379 Sep 26 13:55 testData.txt
root@ser:~/cr#
```

Running the software

```
docker run --network none -it container bash

## enters the container and runs shell



/bin/dec /R_script/model.enc /R_script/testData.txt
```

"Ordinary run" output

```
Loading required package: gplots
KernSmooth 2.23 loaded
Copyright M. P. Wand 1997-2009

Attaching package: 'gplots'

The following object is masked from 'package:stats':

    lowess

[1] 0.745098
[1] 0.8431373
[1] 0.8431373
[1] 0.8823529
[1] "------------------------------------------------------------"
[1] 0.9610895
[1] 0.8823529
[1] 0.9697624
```

Let's now investigate the binary

```
file /bin/dec
/bin/dec: ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.24,
BuildID[sha1]=d172ca9d463deadbeefa77794a40e5b11b31d4a8818d,
not stripped
```

The two sections highlighted in bold are potentially useful

These days, you'd likely feed it into Ghidra or Radare2

But no, we're going to do it the hard way

Quick look at ldd

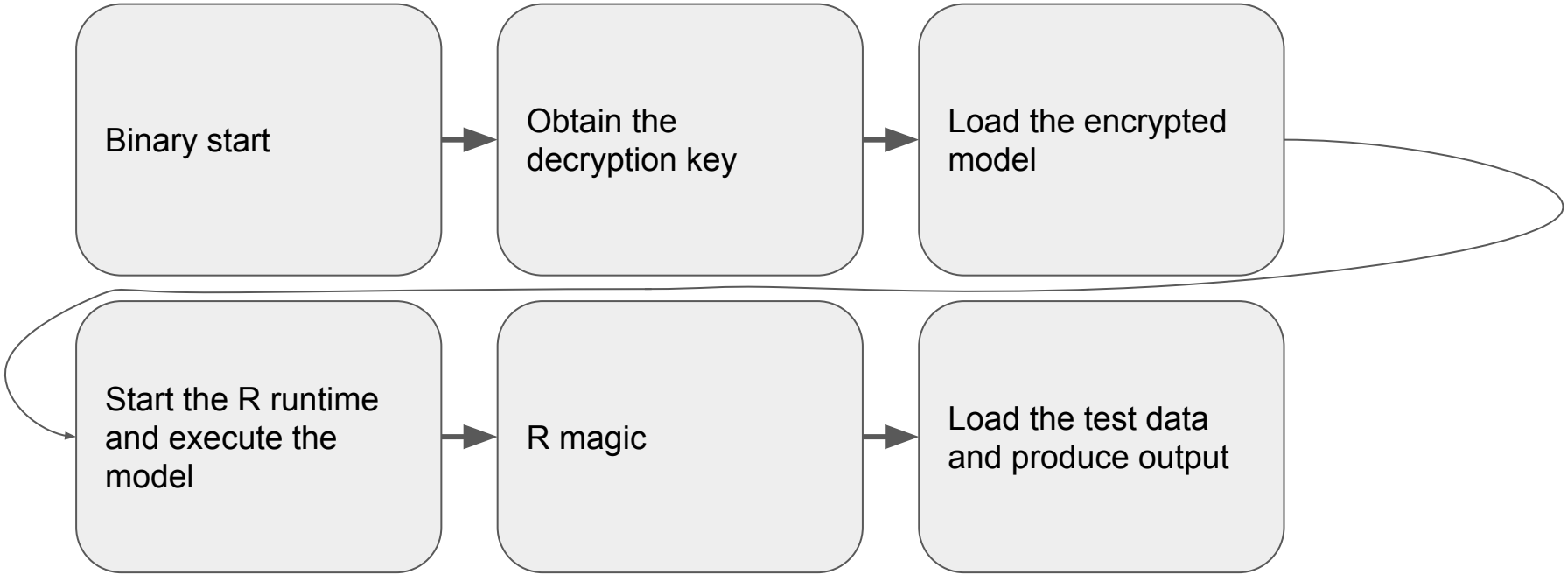
```
root@b2183364e36a:~# !ldd
ldd /bin/dec
        linux-vdso.so.1 =>  (0x00007ffd18f6f000)
        /usr/lib/x86_64-linux-gnu/libssl.so (0x00007fef81a24000)
        libR.so => /usr/lib/libR.so (0x00007fef814d4000)
        libcrypto.so.1.0.0 => /lib/x86_64-linux-gnu/libcrypto.so.1.0.0 (0x00007fef810f8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fef80d2f000)
        libblas.so.3 => /usr/lib/libblas.so.3 (0x00007fef80ab2000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fef807ac000)
        libreadline.so.6 => /lib/x86_64-linux-gnu/libreadline.so.6 (0x00007fef80566000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fef80328000)
        liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007fef80106000)
        libbz2.so.1.0 => /lib/x86_64-linux-gnu/libbz2.so.1.0 (0x00007fef7fef6000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007fef7fcdd000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fef7fad5000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fef7f8d1000)
        libgomp.so.1 => /usr/lib/x86_64-linux-gnu/libgomp.so.1 (0x00007fef7f6c2000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fef7f4a4000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fef81c83000)
        libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007fef7f27b000)
root@b2183364e36a:~#
```

Look at objdump (abbrev)

```
0000000000401872 <decrypt>:
  401872:       55                      push   %rbp
  401873:       48 89 e5                mov    %rsp,%rbp
  401876:       48 83 ec 40             sub    $0x40,%rsp
  40187a:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40187e:       89 75 e4                mov    %esi,-0x1c(%rbp)
  401881:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
  401885:       48 89 4d d0             mov    %rcx,-0x30(%rbp)
  401889:       4c 89 45 c8             mov    %r8,-0x38(%rbp)
  40188d:       e8 5e fb ff ff          callq  4013f0 <EVP_CIPHER_CTX_new@plt>
  401892:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  401896:       48 83 7d f8 00          cmpq   $0x0,-0x8(%rbp)
  40189b:       75 0a                   jne    4018a7 <decrypt+0x35>
  40189d:       b8 00 00 00 00          mov    $0x0,%eax
  4018a2:       e8 b0 00 00 00          callq  401957 <handleErrors>
  4018a7:       e8 e4 fc ff ff          callq  401590 <EVP_aes_256_cbc@plt>
```

The binary uses openssl to facilitate the decryption, and now we know which functions it's calling

(suspected) program flow



How about a very simple OpenSSL "attack vector" ?

The simplest attack vector would be to try and reimplement a few of the juicy OpenSSL functions and LD_PRELOAD them to see if that works. We can also set breakpoints (and use GDB to literally print internal variables). Let's take a look at one of our "new" OpenSSL functions:

```
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl,
                      const unsigned char *in, int inl);
........
    if (ctx->final_used) {
        if (((PTRDIFF_T)out == (PTRDIFF_T)in)
            || is_partially_overlapping(out, in, b)) {
            EVPerr(EVP_F_EVP_DECRYPTUPDATE, EVP_R_PARTIALLY_OVERLAPPING);
            return 0;
        }
        memcpy(out, ctx->final, b);
        out += b;
        fix_len = 1;
    } else
        fix_len = 0;
........
        printf("\n\n%s\n\n",out);   //// this line actually prints decrypted contents (it's ran twice,
once for the private key and then decrypted R script)            //// in my OpenSSL there were many more
such printf statements
```

Output (abridged) of the augmented OpenSSL version

```
Initializing
 aa 7f 5b 29 c8  d 10 1a ::  ec 31 d6 15 a0 43 a7 ce
cipher: AES-256-CBC
EVP SIZE: 8: Key length: 32: update:
IN:
 36 52 1d 50 d8 41 b5 f9
update:  ___


-----BEGIN RSA PRIVATE KEY-----
MIIEowIBOLtCrXQWKyeLBt8bzSB8bebeb1xCMJ+nF73G6ZbN+wq7D0RG123451CkFh
LfyZG6PxnzCFa6e7GRZxxxH4CK4D4Yx5UP3RC0Nxxx0ubc/NXr61d9O3EgOEaxm73X
guH1F8JbOLtCrXQWKyeLBt8bzSB8dI1RxRy/cEom16ghzFn1ontvoL2ZvdrCOGaO
...... (no, this is not a complete private key ;) it has been redacted)
```

Output, continued (the R script portion)

```
Initializing
 30 1f 45 24 14 89 a7 65 ::  40 97 f3 b8 91 5b 3e dc
cipher: AES-256-CBC
EVP SIZE: 8: Key length: 32: update:
IN:
 ec c3 70 d9 72 b1 ca 70
update:  ___
```

```
library(FNN)
library(ROCR)
setwd(".")
rm(list=ls())

qyhUbSmote <- function(inputData, minorLabel, geneNum, numNeighbours, rescaleSize){
```

////The output of the R script itself is irrelevant

Oh, so we "hacked" the software by putting printf's into OpenSSL library

Yay.

That was the *hard* way.

Let's think about even easier way to get what we need...

Cracking like it's '88 (or anytime until about '95)

Who remembers the old freezer cartridges that 8-bit computers used to use?

Show of hands?

Back in the old good days, one could freeze a machine in time and poke around its memory, also being able to single-step the CPU

Can we do that on a modern PC ?

Sure, even though modern hardware won't help you (can't freeze DRAM, not without actually cryo-freezing it)

The biggest challenge is going to be *not HOW, but at WHICH point in time to freeze the execution.* It’s not easy to time things, given how fast modern computers are


ACTION REPLAY
FREEZE BUTTON

How can we "freeze" our program at the point of interest?

Notice how we ran it with 2 arguments? (`model.enc testData.txt`)

The first was the filename of the encrypted script (likely read by the binary AFTER it has the decryption keys in memory :D)

The second was the test data (read by R runtime during execution of the script)

Bet that freezing the execution BEFORE loading the encrypted model might have the keys in memory by then

Bummer, our machine does not have a "pause" or "freeze" button

If only we had a "freeze" button...

Let's get back to basics. Is there any way to effectively freeze execution of a binary? hint hint, a binary that loads another file along the way.

Would you like to hear about FIFO’s ?

FIFO looks like a normal file, but any `read()` system call will wait indefinitely until data shows up in the FIFO.

So let's point our binary at a fifo and make it wait... forever… by never giving it any data...

```
mkfifo our_fifo.foo
```

```
./dec our_fifo.foo testData.txt
```

(We don't need our custom OpenSSL here)

The binary is now effectively "paused".

Let's access its memory. First get the pid (`ps`) and then run `gcore <pid>`

Yay, now we have the dump of the memory...

This is a few screens down in the corefile, opened using ordinary "less" command

```
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@<C1>^P^@^@^@^@^@^@library(FN
library(ROCR)
setwd(".")
rm(list=ls())

qyhUbSmote <- function(inputData, minorLabel, geneNum, numNeighbours, rescaleSize){
  minorLoc <- which(inputData[, ncol(inputData)]==minorLabel)

  minorData <- inputData[minorLoc, ]

  selectMinLoc <- sample(1:length(minorLoc), geneNum, replace = T)
  selectMinData <- minorData[selectMinLoc, ]
```

R source code starts here :)

Nice huh?

We've just retrieved the decrypted source of the R script from the coredump.

Didn't even need the private key shown earlier.

Could this have been prevented ?

- The creator could have rolled their own crypto, we all know how great that is ;) (seriously, as bad as it seems, the OpenSSL attack wouldn't have been easy. At the least, obfuscating OpenSSL, modifying function signatures would have made it hard to replicate)
- Use different OpenSSL library, perhaps closed source (harder to mimic for LD_PRELOAD)
- Realistically, escrowing the keys using an online service or hardware key might be one way
- Obfuscating the R code would be another option (but it would still be available in the coredump)
- Preventing the simple "FIFO attack" could have been done by embedding the encrypted model in the binary and/or loading the model early on, before getting the keys

Could this have been prevented ?

- Detect FIFO / ensure that the files are actually files (simple stat() would help), or even better - digitally sign + checksum any files that would be loaded at startup
- Statically link libraries would have made the LD_PRELOAD harder (might be hard to statically link OpenSSL though, I've never tried)
- There are ways to "disable" LD_PRELOAD, either by using different libc (such as musl) - the challenge would be ensuring the code can still run at customer's site
- Have the binary inspect every loaded shared object (via callbacks) and looking for unexpected libraries (not great, but it would certainly make it harder for script kiddies)
- There are several "anti-forensics" techniques of scrambling the memory, often used in malware/viruses to reduce risk of detection. Most often manipulating Page Table Entries and/or Memory Area Structures in kernel

Recap

- Yes, admittedly there wasn't much thought put into protecting this particular application
- Most of the preventive measures I mentioned are very reactive and protect from a specific vector
- Sometimes, most primitive methods work well, printf() and FIFO ftf!

ありがとうございました