Raytheon

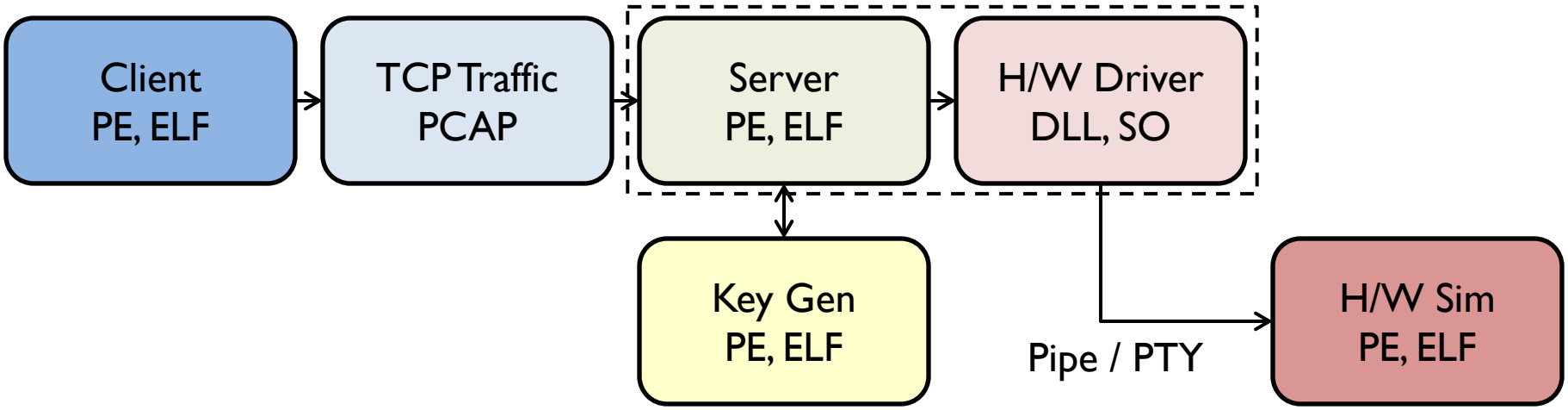
NSA Codebreakers Challenge

Bernie Lampe

Overview

- Yearly NSA recruiting CTF-like event
- Sept 5, 2016 to Dec 31, 2016
- Six levels that build on one another
- Progress to next level sequentially
- Exponentially more difficult levels
- Motivating story is defeating IED software
  - To stop terrorists of course
- Linux guy so this will have a Linux slant

Game Map – 6 Levels



- Binary Reversing
- Network Traffic Analysis
- Public/Private Key Pair
- Crypto Cracking
- Static/Dynamic Analysis
- Malware Reversing

Client

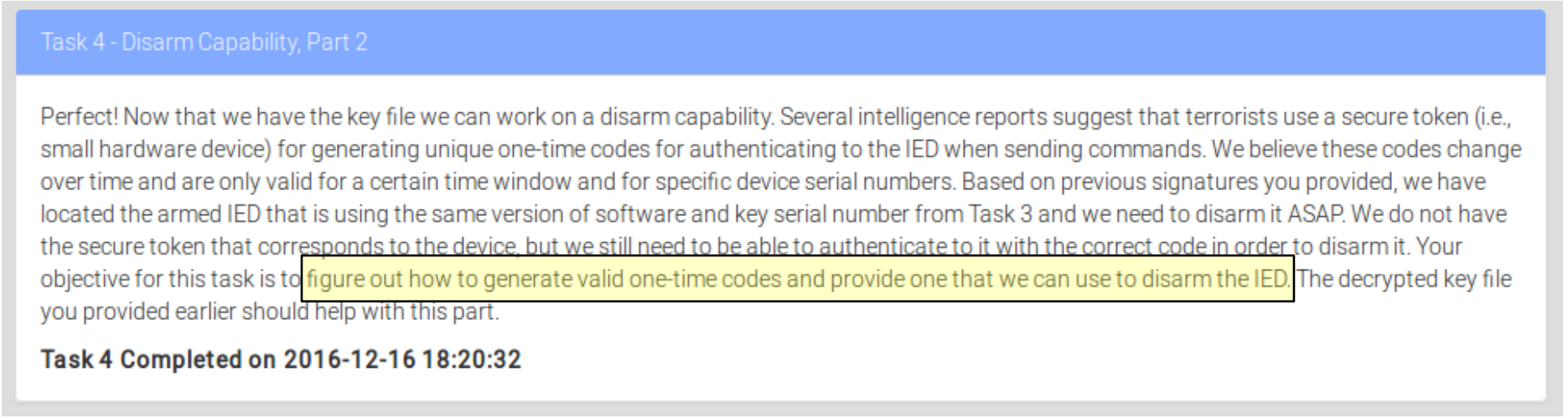
- The Client Program
  - Connecting to the server obtained in later stage

```
connecting to host 127.0.0.1
got serverhello
SERVERHELLO signature correct!
Remote OTP label is 767936614
> ?
invalid command specified. valid commands are:
exit
arm
disarm
trigger
getserial
getstate
raw <command_id> [<arg_data_in_hex>]
> arm
Enter OTP for '767936614':
** invalid OTP! **
>
```

Server

- The Server Program
  - Validates the client signature and commands
  - Computes and validates OTPs
  - Relays commands to the hardware driver
  - Requires an encrypted key file which is keyed to the specific IED serial number
    - IED serial number optionally taken from the env and matched with the decrypted key on startup

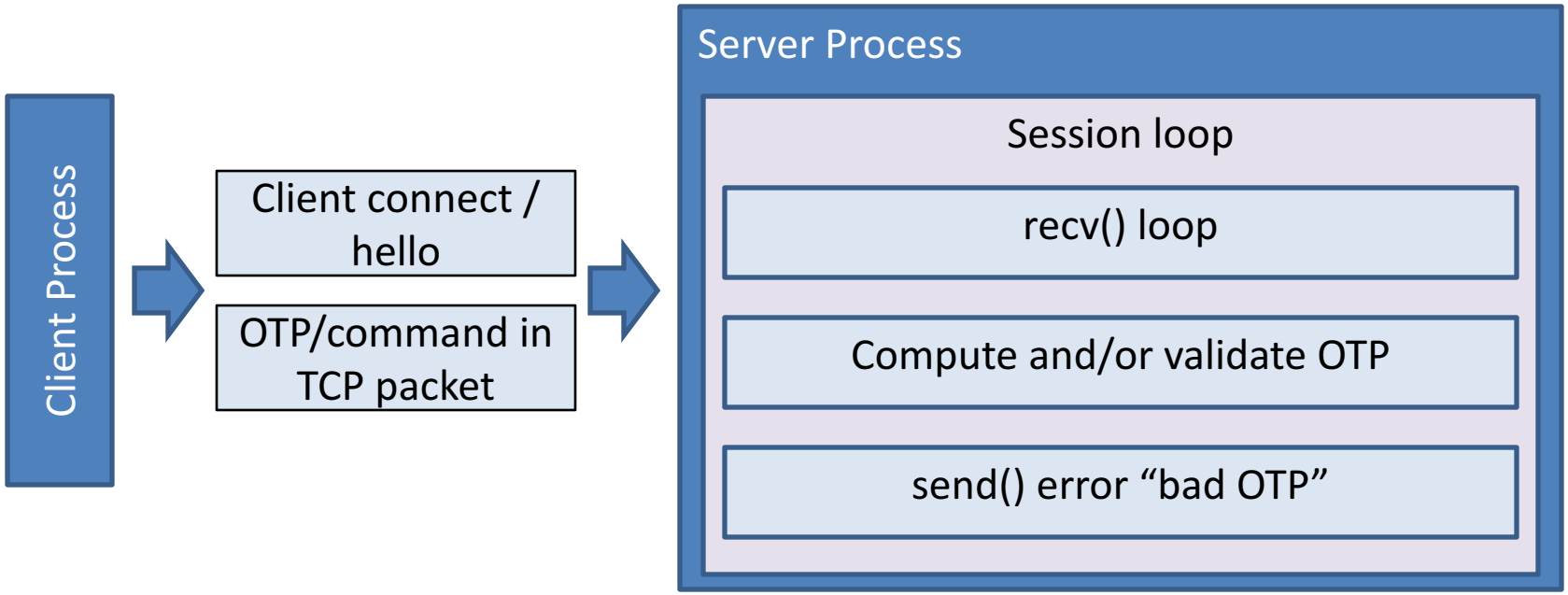
Task 4: Disarm an IED with the key



- Given a valid keyfile and a dummy driver, generate the valid OTP
- OTPs are time based

Task 4 - Hypothesis

- The server is validating OTP's somewhere. Is it computing them?
- Find the validation function and use a debugger to print them?
- Think about how client and server interact to reduce code to look at
  - Build and refine mental model of how server works through iterative dynamic/static reversing/analysis



Task 4 - Solution

- Found the session management loop by breaking on accept()
  - The work for each session is done in its own function

Task 4 - Solution

- Break on recv() calls and get xrefs to this function
  - Try to find when/where the data of interest is read into memory

```
int __cdecl recv_loop(int fd, int a2, int a3)
{
  unsigned int v3; // eax@3
  unsigned int v4; // ebp@3
  int result; // eax@6

  if ( a2 && a3 )
  {
    v3 = 0;
    v4 = 0;
    while ( 1 )
    {
      result = recv(fd, (void *)(a2 + v3), a3 - v3, 0);
      if ( result == 0xFFFFFFFF )
        break;
      if ( !result )
      {
        *__errno_location() = 5;
        return 0xFFFFFFFF;
      }
      v4 += result;
      v3 = v4;
      if ( a3 <= v4 )
        goto LABEL_8;
    }
  }
  else
  {
LABEL_8:
    result = 0;
  }
  return result;
}
```

Task 4 - Solution

- Break on send() call and see which send informs our client of failure
  - When the client reports failure we know we have our send, then work backwards

```
int __cdecl send_loop(int fd, void *buf, size_t n)
{
  size_t v3; // esi@1
  char *v4; // edi@1
  ssize_t v5; // eax@4
  int v6; // ebx@4
  bool v7; // al@4
  int result; // eax@2

  v3 = n;
  v4 = (char *)buf;
  if ( n )
  {
    do
    {
      v5 = send(fd, v4, v3, 0);
      v6 = v5;
      v4 += v5;
      v3 -= v5;
      v7 = v3 != 0;
    }
    while ( v6 != 0xFFFFFFFF && v7 );
    if ( v6 == v3 && v7 )
      *__errno_location() = 5;
    result = 0;
    if ( v6 <= 0 )
      result = v6;
  }
  else
  {
    result = 0;
  }
  return result;
}
```

Task 4 - Solution

- Worked backwards to this conditional

```
}
EVP_MD_CTX_cleanup(&v39);
if ( !memcmp(&s1, &s2, 0x20u) )
{
  if ( check_otp(v36, *(_DWORD *)v23, *(_DWORD *)(v23 + 4), 1, 1, 0) )
  {
    results_in_bad_otp(fd, (int)a4, 3, 0, 0);
  }
  else
  {
    v19 = v38;
```

- Shallow function with complicated math, time(0) call and comparison
  - Set gdb break point at comparison and print numbers
  - Validate that this is the OTP comparison by entering 43690 = 0xAAAA and checking values in regs at bp

```
EVP_DigestSignInit((int)v8, 0, v9, 0, v10);
EVP_DigestUpdate(v8);
v14 = 0;
v11 = EVP_DigestSignFinal(v8, &v17, &v14);
EVP_MD_CTX_destroy(v8);
if ( v11 != 1 || v14 != 0x14 )
  return 0xFFFFFFFF;
if ( a1 == (((*((_BYTE *)&v17 + (BYTE3(v21) & 0xF)) << 0x18)
           + *((_BYTE *)&v17 + (BYTE3(v21) & 0xF) + 3)
           + (*((_BYTE *)&v17 + (BYTE3(v21) & 0xF) + 1) << 0x10)
           + (*((_BYTE *)&v17 + (BYTE3(v21) & 0xF) + 2) << 8)) & 0x7FFFFFFFu)
         % 0xF4240 )
  break;
if ( v13 < ++v7 )
  goto LABEL_12;
```

Task 4 - Solution

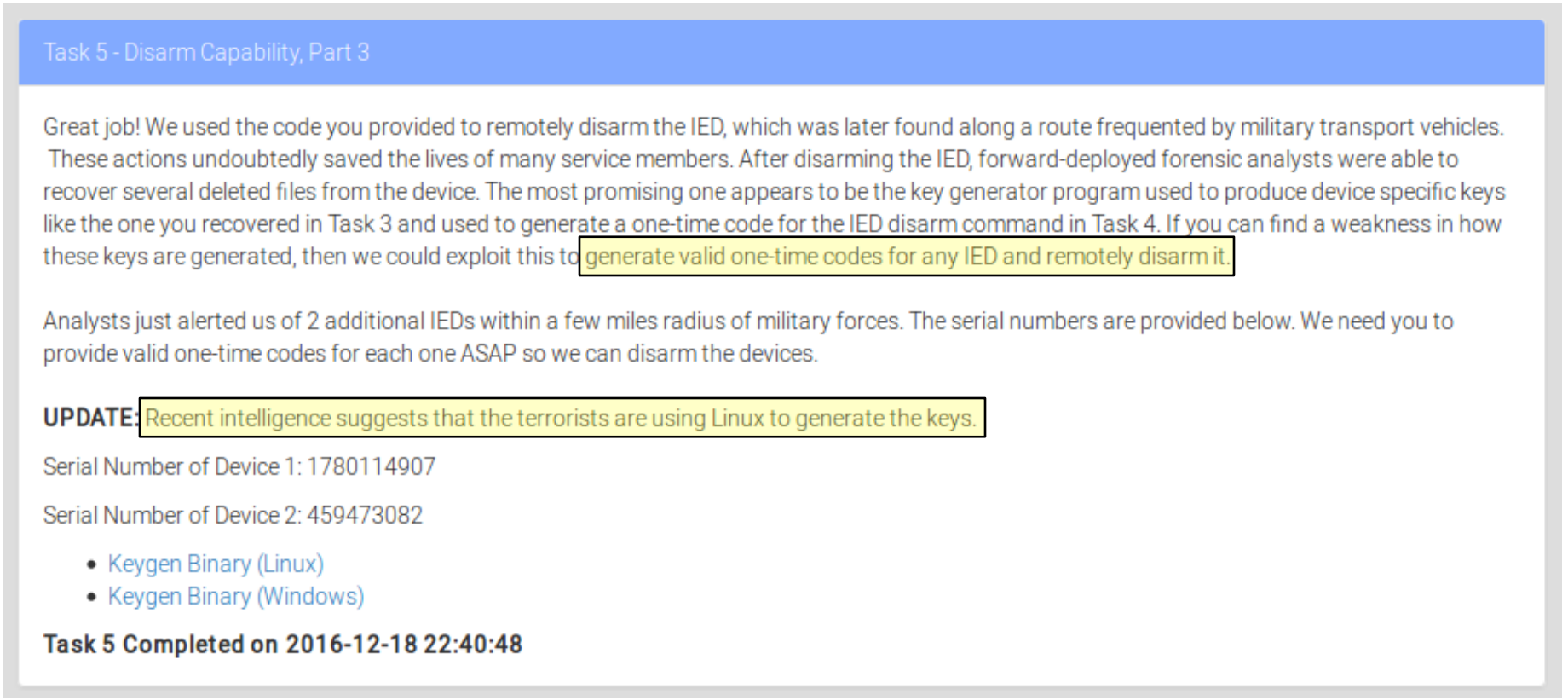
- Too lazy to write a OTP crack program
- Wrote an expect script to drive client and some GDB bp commands

```
#!/usr/bin/expect

spawn ./client
expect ">"
send "arm\n"
expect ":"

for {set i 1} {$i < 999999} {incr i 1} {
    send "$i\n"
    expect {
        "Enter OTP for '767936614'" { }
        "SUCCESS!" { break }
    }
    sleep 0.5
}
```

```
Program received signal SIGINT, Interrupt.
0xf7fdb440 in __kernel_vsyscall ()
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804aedd
        p/x $ecx
        c
(gdb)
```

```
Breakpoint 1, 0x0804aedd in ?? ()
$2981 = 0x353fc

Breakpoint 1, 0x0804aedd in ?? ()
$2982 = 0xe3e3d

Breakpoint 1, 0x0804aedd in ?? ()
$2983 = 0x7b8e0
```

Task 5: Disarm any IED without a key



Task 5 - Disarm Capability, Part 3

Great job! We used the code you provided to remotely disarm the IED, which was later found along a route frequented by military transport vehicles. These actions undoubtedly saved the lives of many service members. After disarming the IED, forward-deployed forensic analysts were able to recover several deleted files from the device. The most promising one appears to be the key generator program used to produce device specific keys like the one you recovered in Task 3 and used to generate a one-time code for the IED disarm command in Task 4. If you can find a weakness in how these keys are generated, then we could exploit this to generate valid one-time codes for any IED and remotely disarm it.

Analysts just alerted us of 2 additional IEDs within a few miles radius of military forces. The serial numbers are provided below. We need you to provide valid one-time codes for each one ASAP so we can disarm the devices.

UPDATE: Recent intelligence suggests that the terrorists are using Linux to generate the keys.

Serial Number of Device 1: 1780114907

Serial Number of Device 2: 459473082

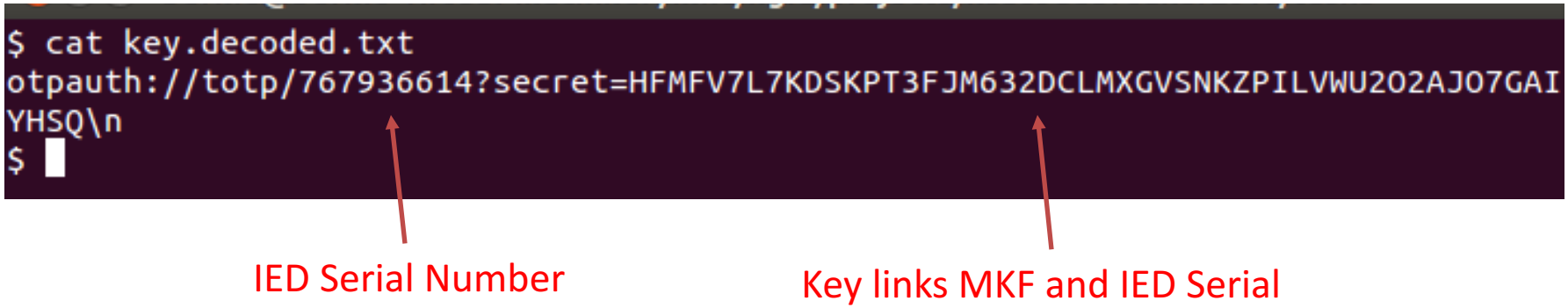
- Keygen Binary (Linux)
- Keygen Binary (Windows)

Task 5 Completed on 2016-12-18 22:40:48

- Without keyfiles, and given the keygen, create valid OTPs
- Two IED serial numbers are given

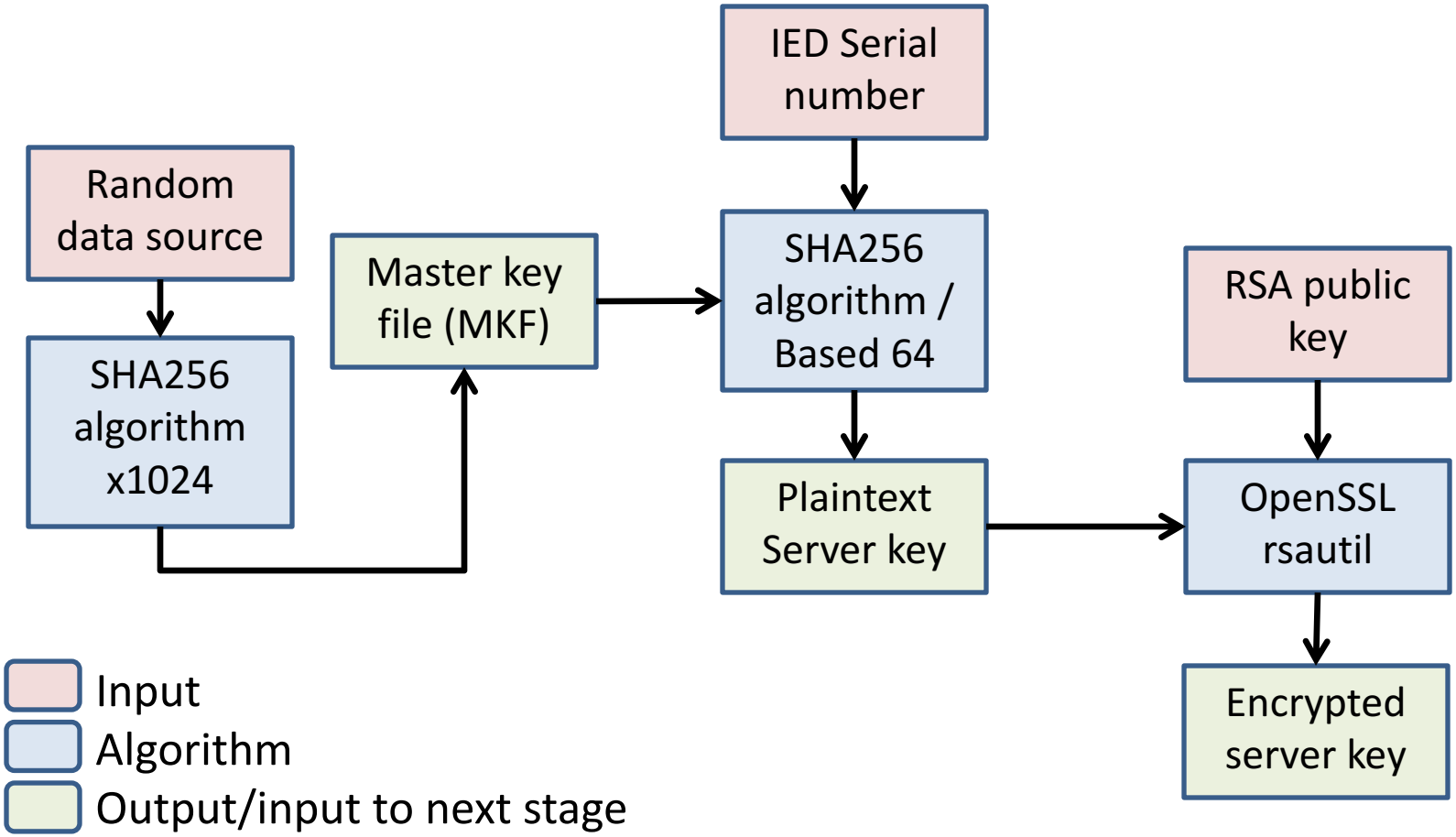
Keygen

- Keygen Program
  - Used to create encrypted server keys for IEDs
  - Must provide the IED serial number
  - Uses a master key file as key seed (we don't have this)
  - Keys are encrypted with pub/priv pair
- Decrypted key



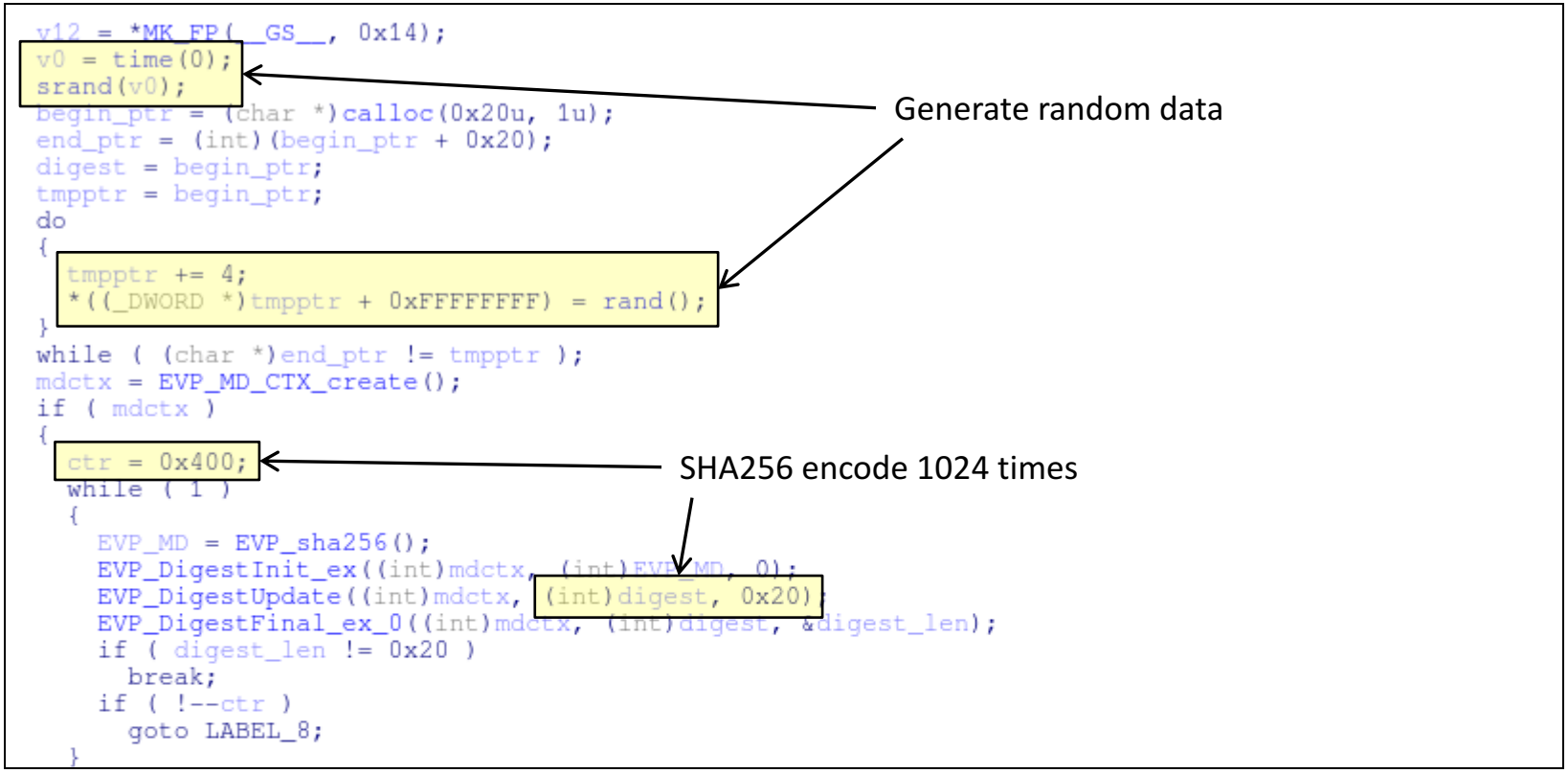
Task 5 - Solution

- The key generating infrastructure



Task 5 - Solution

- Reversing the keygen binary
  - Added all the symbols used from OpenSSL
  - Found the master key generate function



Task 5 - Solution

- Reversing the keygen binary
  - Traced the master file data and input serial number through the program

```
  mdctx = EVP_MD_CTX_create();
  if ( !mdctx )
  {
    __fprintf_chk(stderr, 1, (int)"ERROR: unable to create MD_CTX\n");
    goto LABEL_8;
  }
  EVP_MD = EVP_sha256();
  pkey = EVP_PKEY_new_mac_key(0x357, 0, (int)filedata_2, 0x20);// first parameter == 0x357 == 855 = EVP_PKEY_HMAC
  EVP_DigestSignInit((int)mdctx, 0, (int)EVP_MD, 0, (int)pkey);
  EVP_DigestUpdate((int)mdctx, (int)&input_serial_no, 4);
  EVP_DigestFinal_ex((int)mdctx, (int)digest, (int)&digest_len);
  EVP_MD_CTX_destroy(mdctx);
  if ( digest_len != 0x20 )
  {
    __fprintf_chk(stderr, 1, (int)"ERROR: invalid subkey size\n");
LABEL_8:
    result = 1;
    goto LABEL_9;
  }
  digest_1 = digest;
  v13 = 0x20;
  __snprintf_chk(&v15, 0xC, 1, 0xC, "%d");
  v14 = &v15;
  key_to_url((int)&digest_1);
  __snprintf_chk(filename, 0x20, 1, 0x20, "%d.key", input_serial_no);
  fd = fopen(filename, "wb");
```

Task 5 - Solution

- We have all inputs except the seed to generate the random data
  - The seed is generated with time(0)
  - Assume all IEDs use same MKF

```
v12 = *MK_FP(__GS__, 0x14);
v0 = time(0);
srand(v0);
begin_ptr = (char *)calloc(0x20u, 1u);
end_ptr = (int)(begin_ptr + 0x20);
```

- Brute force algorithm to find time when the IED MKF was generated
  - Override srand(seed) to increment through the seeds in keygen
  - Run keygen –g to make a new MKF on each iteration
  - Use the serial / server key file from task 3 as stopping criteria
  - When we generate that file, we know we are done
  - Narrow down the range of seeds to search by starting with the day before contest
    - Sep 4, 2016

Task 5 - Solution

- Simple shell script to brute force the seed along with LD_PRELOAD

```
#!/bin/bash

i=1472947200 # sep 4 00:00:00 GMT
while true
do
    if [[ $i -ge 1475193600 ]]
    then
        echo "FAILED!!!!"
        exit 1
    fi

    LD_PRELOAD=./mysrand.so SEED="$i" ./keygen -g -k 767936614 -o t

    diff key.decoded.txt 767936614.key > /dev/null

    if [[ $? -eq 0 ]]
    then
        echo "FOUND IT SEED = $i!"
        exit 0
    fi

    i=$((i+1))
done
```

```
void srand(unsigned int seed)
{
    void (*orig_srand)(unsigned int);
    orig_srand = dlsym(RTLD_NEXT, "srand");

    unsigned int new_seed = atoi(getenv("SEED"));
    if (new_seed % 1000 == 0) {
        printf("seed = %d\n", new_seed);
    }

    return (*orig_srand)(new_seed);
}
```

Task 5 - Solution

- Found it in about 10 minutes

```
FOUND IT SEED = 1473013869!

Sun 04 Sep 2016 02:31:09 PM EDT GMT-4:00 DST
```

- Generate two keys, one for each serial number
- Proceed just like in task 3 to print the OTPs
  - Run the servers/client drivers simultaneously using expect script
- Two instances of server on same machine
  - Ports 41266 / 8080 conflict
- Modify the client and server binaries to bind to different ports
  - My favorite binary editor is bvi

Task 6: Permanently disable any IED

Task 6 - Hardware Destruction

Ideally we would like to go one step further and remotely render the IEDs completely inoperable. Given that we may not be able to find and recover disarmed IEDs before the terrorists reclaim them, it would be much better if we could ensure that the IEDs never work again (without actually triggering an explosion and potentially causing civilian causalities.) To that end, we had our hardware reverse engineers study the hardware of a captured IED and develop a hardware simulator to accurately model its operations. This enables more analysts across the enterprise to test and evaluate the system without requiring access to one of the very few captured devices. During development of the hardware simulator, they discovered that if we can somehow cause the hardware to be triggered without being armed then it will "brick" the hardware. This may be our best shot at having a remote disabling capability if we can find a vulnerability to exploit the trigger-before-arm weakness.

From the captured IED we also recovered the hardware driver that contains all the logic for arming/disarming/triggering the actual IED hardware (i.e., unlike the dummy driver that had these functioned stubbed out.) Using this driver with the hardware simulator and IED software provides the most accurate model of the IED hardware and software stack available. The driver appears to contain additional protections that might make analyzing it more difficult, so be on the lookout for code that might be trying to detect dynamic analysis tools.

- hwsim Binary (Linux)
- Real Driver (Linux)
- hwsim Binary (Windows)
- Real Driver (Windows)

Once you have accomplished this task, upload your command file using the form below.

Task 6 Completed on 2017-01-16 15:34:29

- Write client program to cause trigger before arm in hwsim

Client

- The Client Commands

```
connecting to host 127.0.0.1
got serverhello
SERVERHELLO signature correct!
Remote OTP label is 767936614
> ?
invalid command specified. valid commands are:
exit
arm
disarm
trigger
getserial
getstate
raw <command_id> [<arg_data_in_hex>]
> arm
Enter OTP for '767936614':
** invalid OTP! **
>
```

Driver

- The Driver Shared Object
  - In charge of the hardware state machine
    - Shouldn't issue "trigger" command if not in "armed" state
  - Knows how to run the hardware
  - Lots of anti-debugging and anti-reversing here

HW Sim

- The Hardware Sim Program
  - Accepts commands over serial device from driver
  - Simple state machine
  - The story line says this program was constructed by reversing a real IED chipset
  - You can ”brick” the chipset if you trigger w/o arm

Task 6 - Solution

- Goal is to write a client script that will cause hwsim to brick
  - Trigger without arming
  - No OTPs in this level
- Reversing libdriver.so
  - Too much reversing to take you all the way through my thought process
  - Detail the highlights of reversing, anti-reversing, anti-debugging and the interpreter
- Turned out to be academic examples of malware
  - disassembly obstacles
  - red herrings
  - two debugger checks
  - pointer obfuscation via XOR'ing
  - SEH (ported to Linux) to make code flow hard to follow
  - custom interpreter / virtual machine to do the actual work
  - run length encoded interpreter program
  - subtle bug in interpreter

Task 6 - Solution

- Disassembly obstacles
  - Classic jump to the middle of another instruction and put in junk instruction bytes
  - Prevents creating of functions

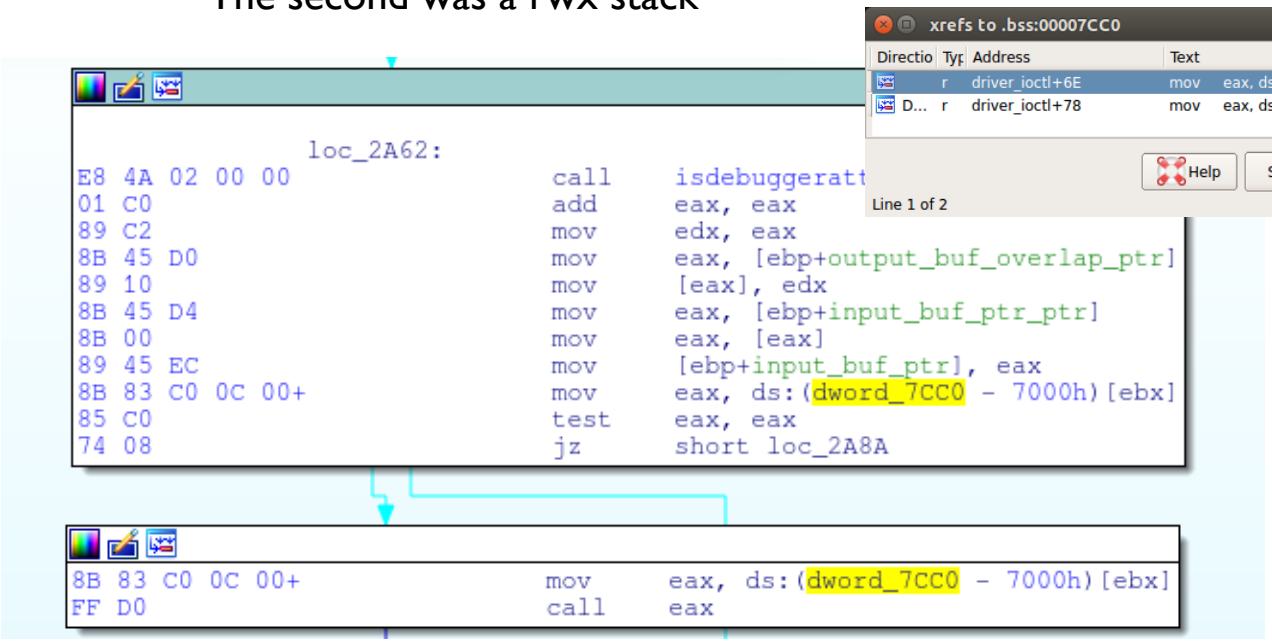
```
.text:00002240                  call    sub_2E2F
.text:00002245                  jmp     near ptr loc_22C6+4
.text:00002245 ; ---------------------------------------------------------------------------
.text:0000224A                  dw 0C744h
.text:0000224C                  dd 0EC832444h, 18858D04h, 50FFFFFFh, 858D046Ah, 0FFFFFF08h
.text:0000224C                  dd 0F861E850h, 0C483FFFFh, 60858910h, 83FFFFFFh, 0FFFF60BDh
.text:0000224C                  dd 2C7501FFh, 0B10E8h, 0B8C28900h, 0
.text:00002284 ; ---------------------------------------------------------------------------
.text:00002284                  sub     eax, edx
.text:00002286                  shl     eax, 8
.text:00002289                  mov     edx, eax
```

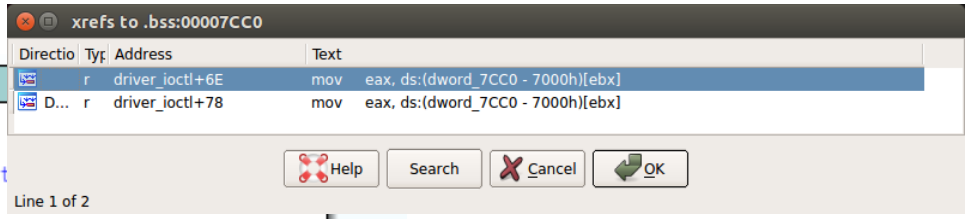
```
.text:00002E27                  leave
.text:00002E28                  retn
.text:00002E29 ; ---------------------------------------------------------------------------
.text:00002E29
.text:00002E29 loc_2E29:                                ; DATA XREF
.text:00002E29                  pop     eax
.text:00002E2A                  add     eax, 6
.text:00002E2D                  jmp     eax
.text:00002E2F
.text:00002E2F ; =============== S U B R O U T I N E =======================
.text:00002E2F
.text:00002E2F
.text:00002E2F sub_2E2F         proc near               ; CODE XREF
.text:00002E2F                                          ; .text:000
.text:00002E2F                  pop     eax
.text:00002E30                  add     eax, 9
.text:00002E33                  jmp     eax
.text:00002E33 sub_2E2F         endp ; sp-analysis failed
.text:00002E33
.text:00002E33 ; ---------------------------------------------------------------------------
.text:00002E35                  align 10h
.text:00002E40
```

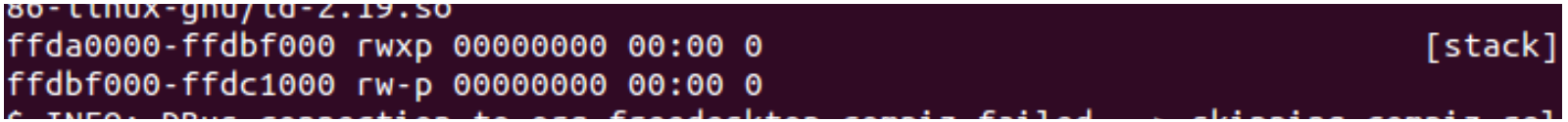
```
.text:00002236                  mov     dword ptr [ebp-0E8h], 0
.text:00002240                  call    sub_2E2F
.text:00002240 ; ---------------------------------------------------------------------------
.text:00002245                  db 0E9h
.text:00002246                  db  80h ;
.text:00002247                  db    0
.text:00002248                  db    0
.text:00002249                  db    0
.text:0000224A                  dd 2444C744h
.text:0000224E                  db  83h ;
.text:0000224F                  db 0ECh ;
.text:00002250                  db 4
.text:00002251 ; ---------------------------------------------------------------------------
.text:00002251                  lea     eax, [ebp-0E8h]
.text:00002257                  push    eax
.text:00002258                  push    4
```

TASK 6 - Solution

- Some red herrings I ran into
  - First one was a global function pointer which was checked then called
  - The second was a rwx stack







Task 6 - Solution

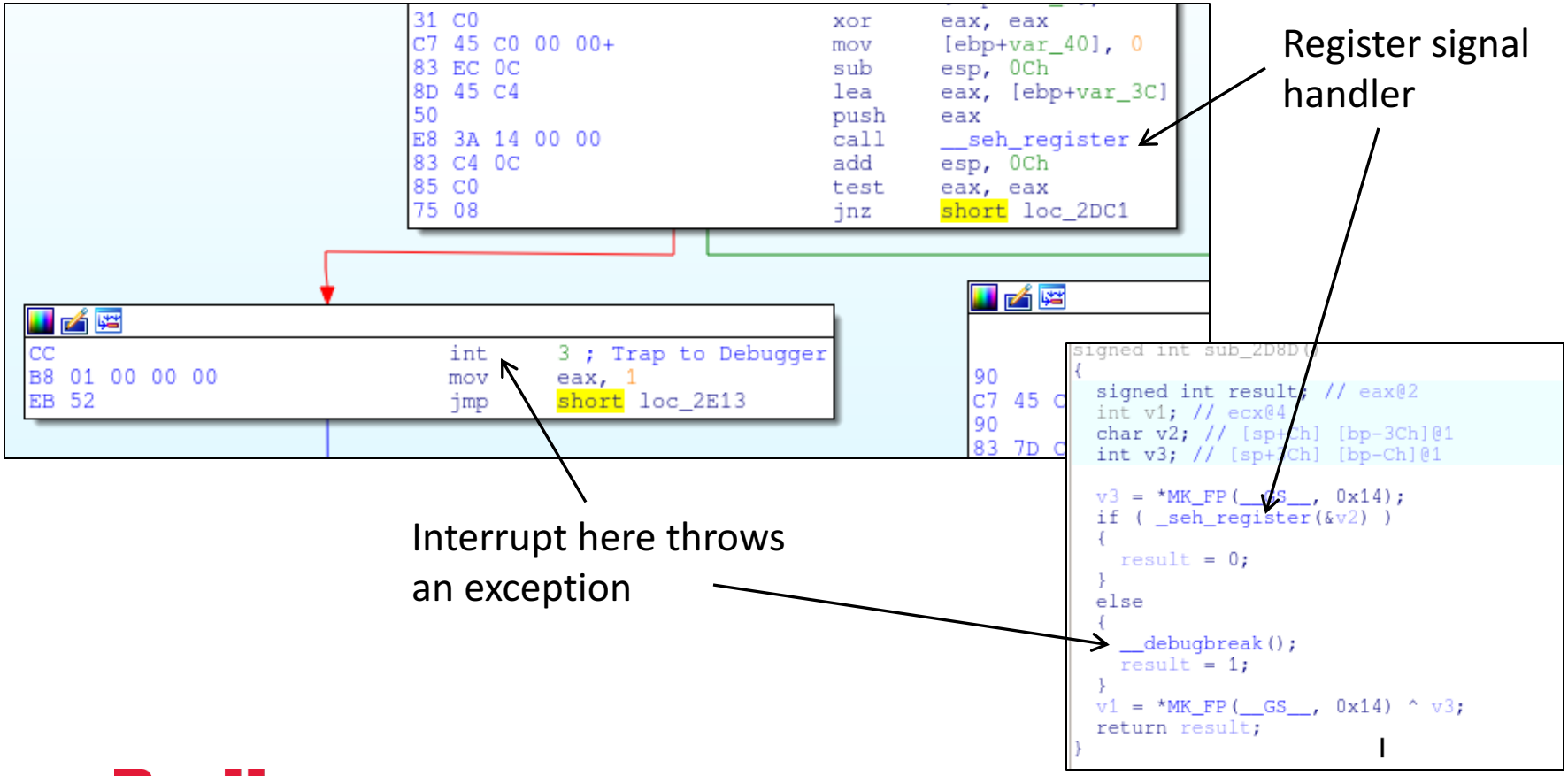
- Debugger check
  - Call to fork() to make a new child process and then tries to ptrace the parent process
  - Easy to overcome with LD_PRELOAD

```
v11 = *MK_FP(__GS__, 0x14);
v9 = fork();
if ( v9 == 0xFFFFFFFF )
{
  result = 0xFFFFFFFF;
}
else
{
  if ( !v9 )
  {
    pid = getppid();
    if ( ptrace(PTRACE_ATTACH, pid, 0, 0) )
    {
      v8 = 1;
    }
    else
    {
      waitpid(pid, 0, 0);
      ptrace(PTRACE_CONT, 0, 0);
      v1 = getppid();
      ptrace(PTRACE_DETACH, v1, 0, 0);
      v8 = 0;
    }
    Exit(v8, v3, v4, v5);
  }
  waitpid(v9, &stat_loc, 0);
  v7 = stat_loc;
  result = (stat_loc & 0xFF00) >> 8;
  v8 = (stat_loc & 0xFF00) >> 8;
```

```
#define _GNU_SOURCE

#include <dlfcn.h>
#include <linux/ptrace.h>
#include <stdio.h>
#include <stdlib.h>

typedef long time_t;

long ptrace(unsigned request, pid_t pid, void *addr, void *data)
{
    return 0;
}
```

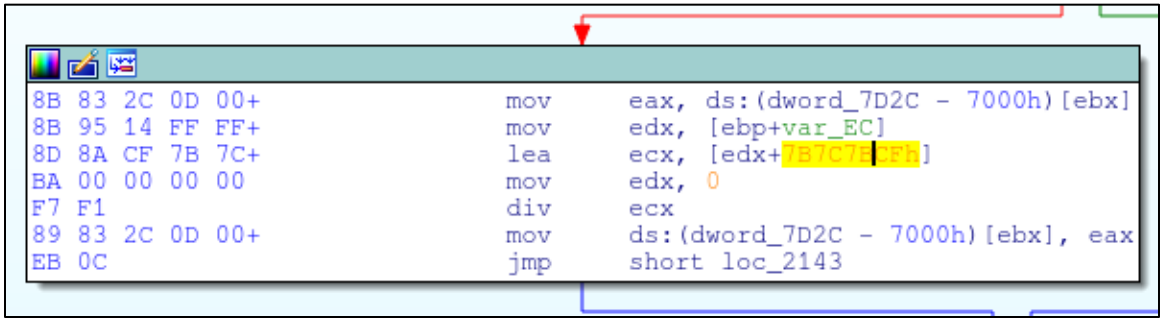
Task 6 - Solution

- Debugger check
  - Ran into another anti-debugger check later on
  - Edit the binary to remove 0xCC and return 0

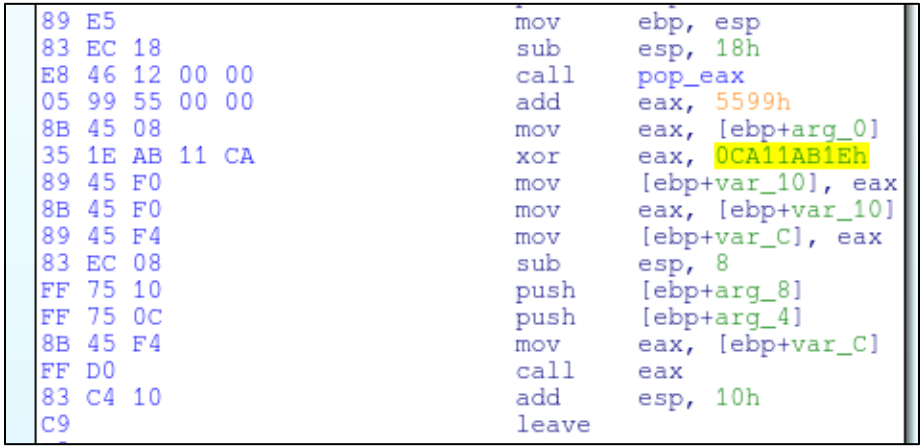


Task 6 - Solution

- Integer overflows to invoke certain control flow



- Function pointer obfuscation via XOR'ing with 0xCA11AB1E



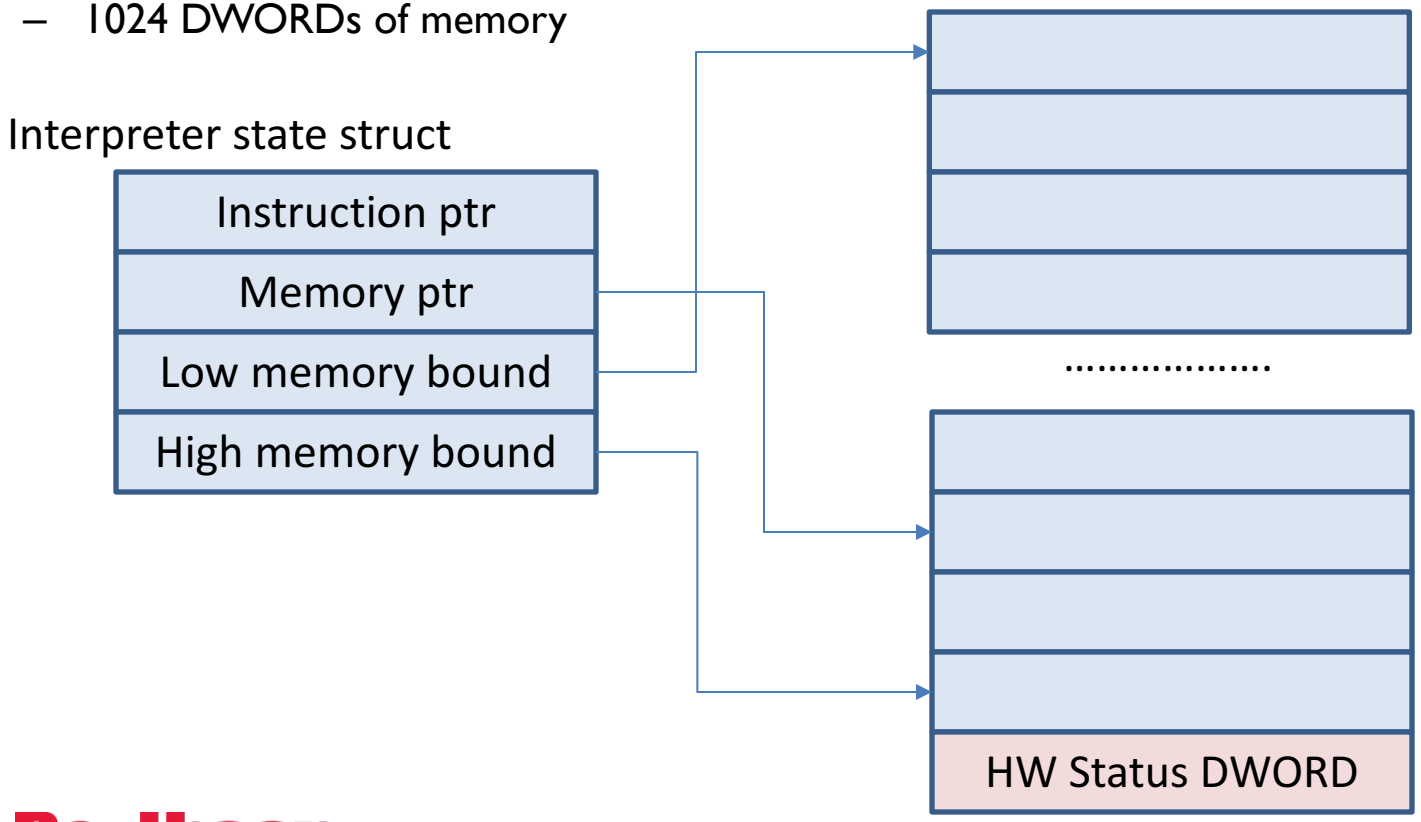
Task 6 - Solution

- Exception handling used to implement logic
  - Thwarts IDAs recursive descent disassembler and obfuscates the logic flow
  - SEH port to Linux by overriding the signal handlers written by Tom Bramer
  - http://www.programmingunlimited.net/siteexec/content.cgi?page=libseh

```
_result = 0;
if ( _seh_register(&__seh_buf1) )
  _result = 2;
else
  dword_7D0C = (int)*input_buf_ptr_ptr;       // deref first argument, to just see i
if ( _result == 3 )
{
  __seh_buf1.state = 1;
  ((void (__cdecl *)(int, struct ___seh_buf *, int, int))__seh_buf1.handler_ptr)(
    __seh_buf1.excinfo_ptr,
    &__seh_buf1,
    __seh_buf1.excinfo_ptr + 0x50,
    __seh_buf1.excinfo_ptr + 0x31C);
}
else if ( _result == 2 )
{
  result = 0xFFFFFFFF;
  goto STACK_CHECK_AND_RTN;
}
call_pop_registration();
dword_7D0C = timediff(dword_7D0C);            // zero out the return
v19 = 0;
if ( _seh_register(&__seh_buf1) )
  v19 = 2;
else
```

Task 6 - Solution

- Interpreter implements generic read, decode, execute loop
  - Each opcode matches to a handler function
  - The memory pointer is bound checked on each instruction
  - Memory is cleared on each program (except in maintenance mode)
  - 1024 DWORDs of memory



Task 6 - Solution

- Reversed interpreter opcodes

```
0c => *(m+1) = *m << *(m+1), m++
23 => *(m+1) = *m >> *(m+1), m++
17 => *(m+1) = *m & *(m+1), m++
19 => *(m+1) = *m | *(m+1), m++
57 => *(m+1) = *m ^ *(m+1), m++
60 => *(m+1) = *m + *(m+1), m++
a4 => *(m+1) = *m - *(m+1), m++
bc => *(m+1) = *m * *(m+1), m++
b8 => instr = *m, m++
bb => *m = ~*m
1e => *(m-1) = *(m+1), m--
7d => *(m-1) = *m, m--
74 <dword> => *(m-1) = <dword>, m--
62 <offset byte> => *m = instr, instr+=offset, m--
95 => *m /= *(m+1), *(m+1) = *m % *(m+1)
7a <offset byte> => if *m != 0; then instr+=2 else, instr+=offset
90 => m = m++
a9 => *m = *(m+1), *(m+1) = *m
ab <offset byte> => *(m-1) = *(m + offset), m--
```

Dispatch Overview

```
int vm_mem[0x200];

int ioctl() {
      if (10 bad commands) arm and trigger
      else dispatch()
}

int dispatch() {
       /* check all argument pointers are valid */
       /* if command len % 3 == 0 then fail */
       /* case 1: get the hardware status DWORD */
       /* case 2: get hardware serial in normal mode */
       /* case 3: disarm and change from normal to testing mode */
       /* case 4: change mode from testing to normal mode and reset hardware status DWORD */
       /* case 5: run the vm in testing mode with our instructions */
       /* case 6: run the vm in normal mode and write to hardware (normal path) */
 }
```

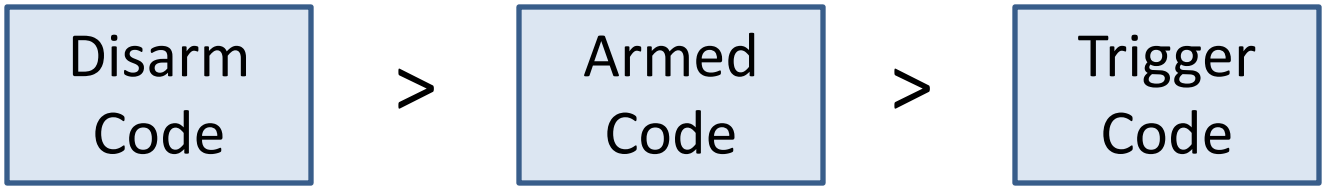
Task 6 - Solution

- Subtle bug in the divide opcode function handlers
  - No bounds checking
  - Allows read divisor and stored remainder to be one DWORD OOB

```
unsigned int __cdecl div_op_no_check(struct vmstate *state)
{
  int mem_ptr; // ecx@1
  unsigned int v2; // esi@1
  unsigned int v3; // ett@2
  unsigned int result; // eax@2

  mem_ptr = state->mem_ptr;
  v2 = *(_DWORD *)(mem_ptr + 4);
  if ( v2 )
  {
    v3 = *(_DWORD *)mem_ptr;
    *(_DWORD *)mem_ptr /= v2;
    *(_DWORD *)(mem_ptr + 4) = v3 % v2;
    result = 0;
    ++state->instr_cnt;
  }
  else
  {
    result = 0xFFFFFFFF;
  }
  return result;
}
```

HW Status DWORD

- 0th byte is disarm code
- 1st byte is armed
- 2nd byte is trigger
- 3rd byte is the command to run
  - It must match one of the other bytes
- Bytes are numerically descending



Task 6 - Solution

- Final answer interpreter program in short hand

```
                                        |   |
| dword | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | y |
                              ^         |   |
0) advance m, m=p8
1) put 0x10 into p7, m=p7
2) put 0x0000ff00 -> p6, m=p6
3) copy dword -> p5, m=p5
4) and p6 = p5 & p6, m=p6
5) shift p7 = p6 << p7, m=p7
6) add p8 = p7 + p8, m=p8
7) decrement m by 6 positions, m=p2
8) put 0x00ffffff into p1, m=p1
9) decrement m, m=p0
10) and p1 = p0 & p1, m=p1
11) advance m 7 positions, m=p8
12) copy p1 -> p7, m=p7
13) or p8 = p7 | p8, m=p8
14) div p8 = p8 / y, y=p8 % y
15) exit 00
```

Task 6 - Solution

- Final answer
  - First 5 commands run the interpreter legitimately and leave state data in memory
  - 6th command changes to maintenance mode so we can run interpreter with our program
  - 7th command is interpreter program that does OOB write
  - Remaining commands fail and trigger an automatic arm-trigger mode

```
getserial
getstate
arm
disarm
trigger
raw -2071755727
raw -2073459836 907410000000 7400ff0000abfa170c60abffabffabffabffabffabff74ffffff
00abff17909090909090abf9199500
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
raw 11 1111
~
```

Game Review

- How does it compare to real life?
  - Pros
    - Demonstration of practical skills and tools
    - Good use of red herrings and false leads
  - Cons
    - Real problems have many more shared objects
      - Much bigger, less symbols and strings
    - Left out some more advanced memory manipulation tech
      - Heap grooming, shell code writing, return-oriented programming
    - Typical malware uses only a few obfuscation techniques

Letter from Adm. Rogers

Raytheon

Fin

- Questions