Give Your Brain a Break

SOLVING FLARE-ON 7 'BREAK' CHALLENGE

Dmitriy Beryoza (0xd13a)

Slides: https://github.com/0xd13a/presentations

Who am I?

Dmitriy Beryoza

@0xd13a on HackFest Discord/Twitter

https://www.linkedin.com/in/beryozad

VECTRA®
SECURITY THAT THINKS

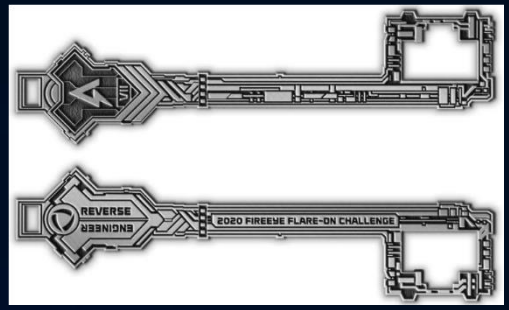
- Senior Security Researcher at *Vectra AI*
- Prior to that - Pentester/Secure Software Development Advocate with X-Force Ethical Hacking Team at *IBM Security*
- 25+ years in software design and development
- Ph.D. in Computer Science, CEH, OSCP
- *Interests*: reverse engineering, secure software development, CTF competitions
- Live in Ottawa, originally from Russia

Agenda

- What is Flare-On?
- Overview of 'break' challenge
- Challenge structure and protection mechanisms
- Solving Stage 1
- Solving Stage 2
- Solving Stage 3
- Wrap-up

Flare-On

- Annual reverse engineering competition (http://flare-on.com/)
- Organized by FireEye
- August-October timeframe
- 10-12 challenges of increasing difficulty
- 40 days to complete them (*this may seem like a lot, but it's not*)
- The toughest RE competition out there *(based on what I've seen - would like to be proven wrong)*
- All finalists get a prize

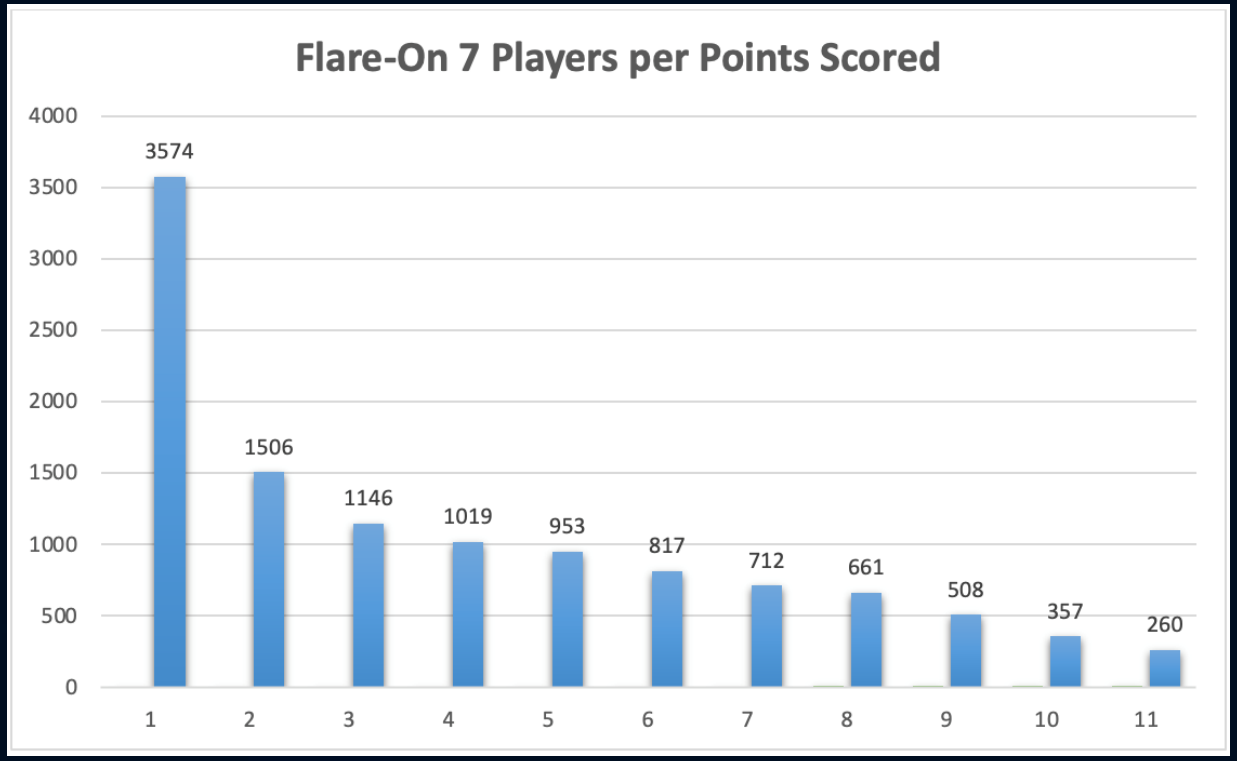


Flare-On (continued)

- You can only solve challenges in order - #2 is enabled when you solve #1
- Challenge flags are all in the form *something@flare-on.com*
- Many challenges are based on real malware samples or malware techniques
- Everything solvable with free software (see https://github.com/fireeye/flare-vm)
- Types of challenges include:
  - All platforms: Windows / macOS / Linux /…
  - Mobile: iOS / Android / Tizen /…
  - P-code: Java / .Net /…
  - Scripting: PowerShell / VBA / Python /…
  - Web: JavaScript / WebAssembly /…
  - 3rd party tools (e.g. AutoIt)
  - Kernel drivers
  - Shellcode
  - Embedded
  - Esoteric (e.g. Subleq, custom VMs)
  - Obfuscation, encryption (including custom), any anti-debug/anti-analysis technique imaginable

Flare-On 7 - 2020 Competition

- 11 challenges
- 5,648 players registered
- 3,574 solved 1st challenge
- 260 finished



- Challenges and solutions: https://www.fireeye.com/blog/threat-research/2020/10/flare-on-7-challenge-solutions.html

'break' Challenge (#10)

- 32-bit Linux executable (1,395 Kb)

```
./break: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=179
3c43108b544ef35f9814b0caafcf76210631c, stripped
```

- On the surface - simple command line challenge, you enter a password and it checks it

```
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one so
ooper easy

please enter a password friend :) test
sorry, but 'sorry i stole your input :)' is not correct
```

- But appearances are deceiving - this part is immediately concerning

Let's Crack It Open

I will use disassembly, cleaned-up decompiled code, and pseudocode - where appropriate

- The challenge code looks too simple to be true:

```
void main_8048D06() {
  char buf[264];

  puts("welcome to the land of sunshine and rainbows!");
  puts("as a reward for getting this far in FLARE-ON, we'v
  putchar(0xA);
  printf("please enter a password friend :) ");
  buf[read(0, buf, 0xFFu) - 1] = 0;
  if ( check_password_8048CDB(buf) )
    printf("hooray! the flag is: %s\n", buf);
  else
    printf("sorry, but '%s' is not correct\n", buf);
  exit(0);
}
```

```
int check_password_8048CDB(char *s1) {
  return strcmp(s1, "sunsh1n3_4nd_r41nb0ws@flare-on.com") == 0;
}
```

- And yet the hardcoded password is not accepted:

```
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one so
ooper easy

please enter a password friend :) sunsh1n3_4nd_r41nb0ws@flare-on.com
sorry, but 'sorry i stole your input :)' is not correct
```

- *What is going on???*

Much More Under the Hood Than Meets the Eye

- It turns out there is a lot going on in the initializer function:

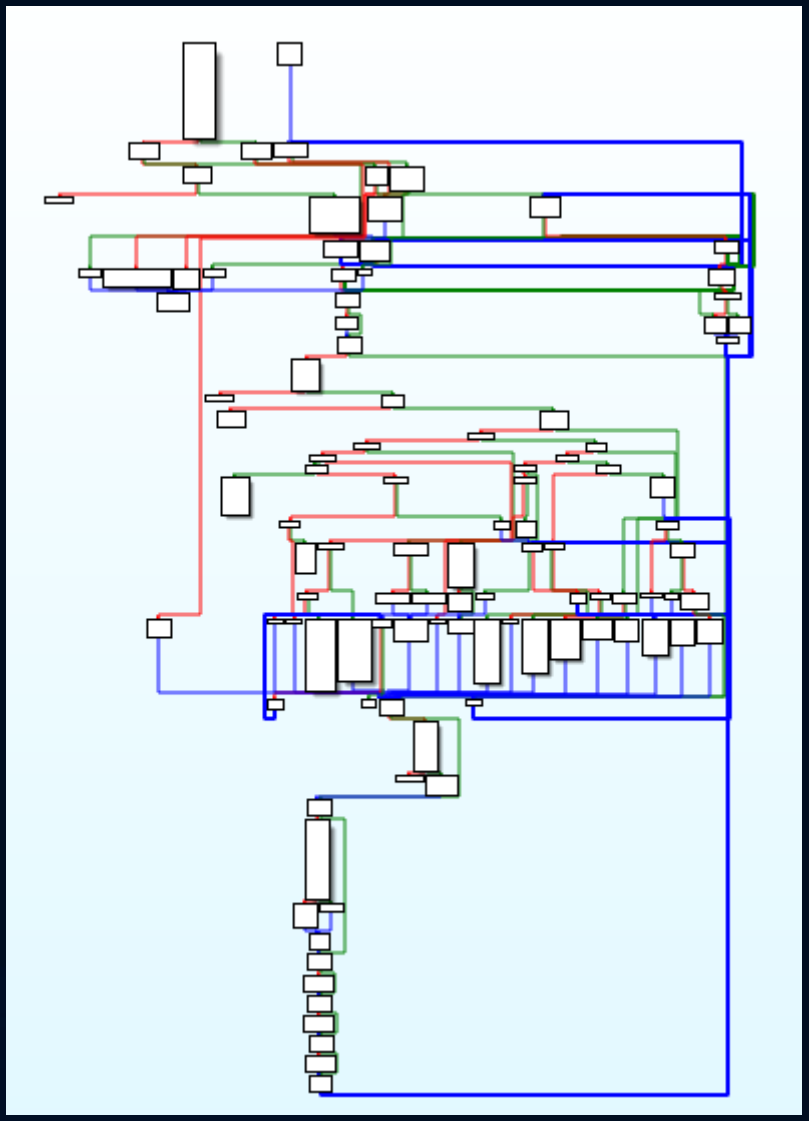
```
                public start
start           proc near
                xor     ebp, ebp
                pop     esi
                mov     ecx, esp
                and     esp, 0FFFFFFF0h
                push    eax
                push    esp             ; stack_end
                push    edx             ; rtld_fini
                push    offset fini     ; fini
                push    offset init     ; init
                push    ecx             ; ubp_av
                push    esi             ; argc
                push    offset main     ; main
                call    __libc_start_main
                hlt
start           endp
```

- One of nested init functions at 0x8048FC5 is interesting, it forks the process and the child enters a big control block:

```
int init_8048FC5() {
  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 2, 0);
  int pid = getpid();
  if (!fork()) {
    main_control_block_80490C4(pid);
    exit(0);
  }
  ...
```

Main Functionality Blocks

- The child control block contains an execution loop that processes and dispatches messages
- At one point another process is spawned (using fork() again) and another big control block is started (function at address 0x8049C9C)
  - This control block also has a big execution loop that processes and dispatches messages
- These 2 big blocks of functionality are what we will concentrate on



Control Mechanisms

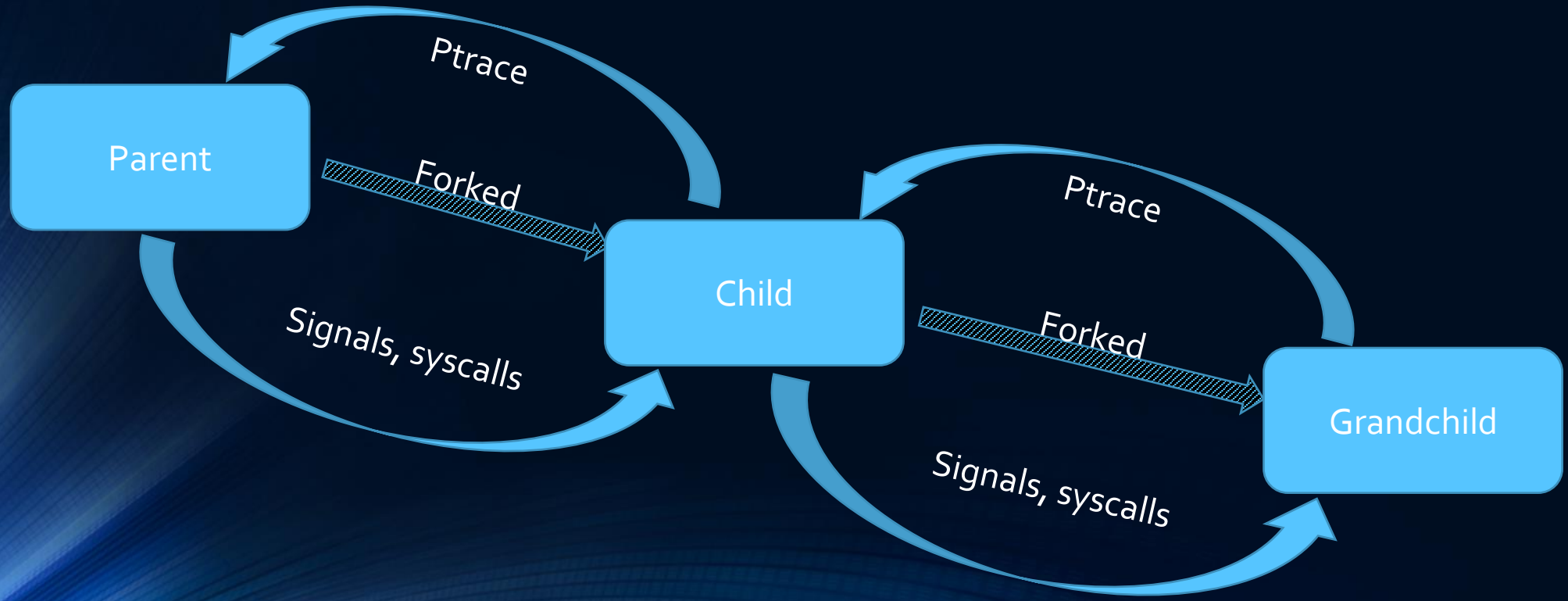
- Inside the main control blocks there are several interesting mechanisms:
  - *Ptrace* - debugging functionality
    - This is what GDB and other tracing and analysis tools use
  - *Signals* - used for RPC communications
    - Most commonly used for handling/communicating extraordinary events (e.g process terminations)
  - *System calls* - invoke standard service functionality from the kernel
    - This challenge overrides standard system calls and gives them completely different purpose
- All of this is bad news, as it makes debugging unfeasible

How Processes Fit Together

- 2 new processes are spawned

```
207690    1966  0  2297  4956    0 Jan18 pts/1    00:00:01          |       \_ /bin/bash
831220  207690  0   929   620    0 22:53 pts/1    00:00:00          |       |   \_ ./break
831221  831220  0   963    64    0 22:53 pts/1    00:00:00          |       |       \_ ./break
831222  831221  0   963    76    0 22:53 pts/1    00:00:00          |       |           \_ ./break
```

- They communicate through ptrace and signals+syscalls



Ptrace

- Several types of ptrace operations are used in the challenge:

Name	Purpose
PTRACE_SETREGS	Sets registers in a debugged process
PTRACE_GETREGS	Gets values from a debugged process
PTRACE_PEEKDATA	Gets a WORD from a debugged process at specified address
PTRACE_POKEDATA	Sets a WORD to a debugged process at specified address
PTRACE_ATTACH	Attach to debugged process
PTRACE_CONT	Continue interrupted process
PTRACE_DENY_ATTACH	Anti-debugging technique

- With ptrace functionality parent processes are controlling the children, triggering the execution
- When children interrupt (by sending a signal or making a system call) - parent exchanges data with them with PEEK/POKE/SETREGS/GETREGS

Signals

- Children use signals to convey information to the parent and trigger functionality in it

Name	Purpose	Caught to
SIGILL	Illegal instruction executed	Trigger the beginning of functionality execution (a special illegal instruction is placed in the code - `0xB0F`)
SIGSEGV	Segmentation fault (memory violation)	Trigger execution of special functionality (by setting parameters and calling the NULL pointer)
SIGALRM	Alarm triggered	Trigger creation of grandchild process
SIGTRAP	Exception occurred (e.g. function executed)	This is be sent when a system call was made, and will be intercepted by a parent
SIGINT	Process interruption requested	Protect against shutdown
SIGTERM	Process termination requested	Protect against shutdown
SIGQUIT	Process exit requested	Protect against shutdown
SIGSTOP	Process stop requested	Protect against shutdown

System calls

- System calls are overridden in the challenge to obfuscate their true nature
- Parent processes use system calls to request specific functionality
- Child processes catch the SIGTRAP signal and performs the actual work
- Here are some of the overridden functions:

Name	Original purpose	Usage in the challenge
execve	Execute OS command	Removes newline from the end of the flag. It is called with parameters "rm -rf --no-preserve-root /" 😱
setpriority	Set priority of the process	Decrypts a string
mlockall	Lock process memory	Counts the number of 1 bits in the parameter
read	Read user input	Reads user flag and replaces it with "sorry I stole…"
… and others		

Decoys and Encryption

- There are several encryption algorithms employed in the challenge
  - From simple XORs to custom encryption
- Lots of decoy values added in order to distract:
  - False flags:

```
"moc.eralf-on@gn1hs1n1f_n0_st4rgn0c",
"elf_0n_a_sh3lf@on-flare.com",
"okay_1_sw34r_th1s_1s_th3_r34l_0ne@no-flare.com",
"sm0l_bin_b1g_h34rt@no-flare.com",
"fake_flag@no-flare.com",
"not_a_fake_flag@no-flare.com",
```

  - Unused shellcode:

```
00 6A 68 68 2F 2F 2F 73  68 2F 62 69 6E 89 E3 68  .jhh///sh/binëph
01 01 01 01 81 34 24 72  69 01 01 31 C9 51 6A 04  .....4$ri..1+Qj.
59 01 E1 51 89 E1 31 D2  6A 0B 58 CD 80 00 00 00  Y.ßQëß1-j.X-Ç...
00 00 00 00 00 00 00 00  00 00 00 00 00 F9 1D 02  .............·..
```

  - Taunts…

```
"This string also has no purpose and again, is here merely to waste your time....",
"This string has no purpose and is merely here to waste your time",
"By reading this string I have successfully stolen around 2 seconds of your life. How does that make you feel?",
"Wasting your time1",
"Wasting your time2",
"Wasting your time3",
```

How Do We Proceed?

- This architecture shows that debugging the code as-is is not possible
- What do we do?

- Static analysis
  - This is unavoidable in many cases, but made harder because debugging is restricted
  - Details of communications between processes have to be fully understood before we proceed
- Selective patching
  - Clear out certain parts of code with NOPs to disable them, or add new functionality

How Do We Proceed? (continued)

- Selective debugging
  - While regular debugging session is not possible, we can still debug pieces of the application
  - We just have to avoid hitting the ptrace invocations, the system calls or the signals (patching helps)
  - Use strategic placement of breakpoints and manually setting the registers.
- Reimplementing some of the code
  - In cases where debugging of isolated pieces of code is complicated, we would have to reimplement them in Python

How Do We Proceed? (continued)

- Building a monitoring scaffolding
  - We can reimplement system calls of interest (*including ptrace*) as a shared library
  - Each call logs the parameters being passed in and result values; we can also dump memory blocks
  - Outside of logging just call the original call

```
int truncate(const char *path, off_t length) {
    printf("### truncate    %s %d\n", path, length);
    int val = ((int (*)(const char *, off_t))dlsym(RTLD_NEXT, "truncate"))(path, length);
    printf("### truncate    returned %p\n", val);
    return val;
}
```

- The library is preloaded before the challenge:

```
export LD_PRELOAD=./over.so; ./break; export LD_PRELOAD=
```

- This gives us a log of calls with all the data, and helps make sense of what's going on
- A lot of logs to go through…

Stage 1

- Armed with described analysis techniques we start analysis of the code that verifies the flag
- This verification is triggered by the invalid instruction and SIGILL signal
- It looks like the flag is being analyzed piece meal

- First section is checked in function located at address 0x8048DCB
- Code takes encryption key that is itself encrypted but is revealed with `nice()` function override
- That key is used to decrypt 16 bytes of hardcoded ciphertext at address 0x81A50EC
- The plaintext is then compared to first 16 bytes of the flag

Stage 1: Decrypting Hardcoded Flag Piece

- The following is a reconstructed piece of code the does the decryption:

```
decryption_setup_804B495(key, decryption_key);
decrypt_804BABC(key, ciphertext_1_81A50EC);  // 2F D4 7F 00
decrypt_804BABC(key, ciphertext_2_81A50F0);  // 98 1C 9C 29
decrypt_804BABC(key, ciphertext_3_81A50F4);  // 3F CE BF B6
decrypt_804BABC(key, ciphertext_4_81A50F8);  // A1 D4 BF 6B
if (!memcmp(flag, ciphertext_1_81A50EC, 0x10u) ) {
  ... // flag portion matched!
```

- Luckily, decryption is happening here before the user-entered data is checked
- We can fool the functionality by doing selective debugging (avoiding forking, ptrace, etc) and jumping around in the code and loading registers manually
- The expected first part of the flag is revealed:

w31c0mE_t0_Th3_1

Stage 2

- Second stage was much more labor-intensive to reverse
- It made much more use of functionality distributed between 3 processes
- I opted for reimplementing the algorithm in Python because this time around the logic actually took the 2nd part of the flag (32 characters), encrypted it, and then compared it to ciphertext at address 0x81A5100
- Because only encryption algorithm was available, I had to build the "inverse" logic to decrypt the data
- The decryption key here is derived as a checksum of string `"This string has no purpose and is merely here to waste your time."`
  - Not everything that looks like a decoy is a decoy…
  - Checksum function is at address 0x804BFED

Stage 2: Decryption Script

- The ciphertext can be found at address 0x81A5100
- `setup()` function is based on code at address 0x804C217
- This is a *key scheduling* routine invoked for every round

```
import struct

flag_part2_ciphertext = bytearray([0x64, 0xA0, 0x60, 0x02,
0xEA, 0x8A, 0x87, 0x7D, 0x6C, 0xE9, 0x7C, 0xE4, 0x82, 0x3F,
0x2D, 0x0C, 0x8C, 0xB7, 0xB5, 0xEB, 0xCF, 0x35, 0x4F,0x42,
0x4F, 0xAD, 0x2B, 0x49, 0x20, 0x28, 0x7C, 0xE0])

keybuf = [0] * 0x1000

def setup(key):
    global keybuf

    val = key
    for i in range(0x10):
        offset = ((i << 8) - (i << 3))
        keybuf[0x1C+offset] = val & 0xffffffff
        keybuf[0x4C+offset] = (val >> 32) & 0xffffffff
        keybuf[0xA4+offset] = (bin(val).count("1") / 2) & 0xffffffff

        bit = val & 1
        val >>= 1

        if bit == 1:
            val ^= 0x9E3779B9C6EF3720
```

Stage 2: Decryption Script

- Decryption is performed for blocks of 8 bytes each
- Decryption key is the one we described earlier (checksum of a string)
- `decrypt()` function is based on code at address 0x804C369
- Here we are just decrypting 32 bytes, but there is more (explained further…)

```
def decrypt(data, idx, key):
    setup(key)
    b = struct.unpack("<L",data[idx+0:idx+4])[0]
    a = struct.unpack("<L",data[idx+4:idx+8])[0]
    for i in range(0x10):

        temp = a

        offset = (((0xf-i) << 8) - ((0xf-i) << 3))

        t1 = (a + keybuf[0x1c+offset]) & 0xffffffff

        t2 = ((t1 >> (keybuf[0xa4+offset] & 0x1F)) |
        (t1 << (-(keybuf[0xa4+offset] & 0x1F) & 0x1F))) & 0xffffffff

        a = b ^ (t2 ^ keybuf[0x4c+offset])
        b = temp

    data[idx+0:idx+4] = struct.pack("<L", a)
    data[idx+4:idx+8] = struct.pack("<L", b)

key = 0x674a1dea4b695809

for i in range(0,0x20,8):
    decrypt(flag_part2_ciphertext, i, key)

print flag_part2_ciphertext
```

Stage 2: Decrypting…

- When we run the script we get the 2nd part of the flag:

w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd

- But the example above is actually using a reduced loop, the real code decrypts much more data - 0x9c40 bytes (40k)

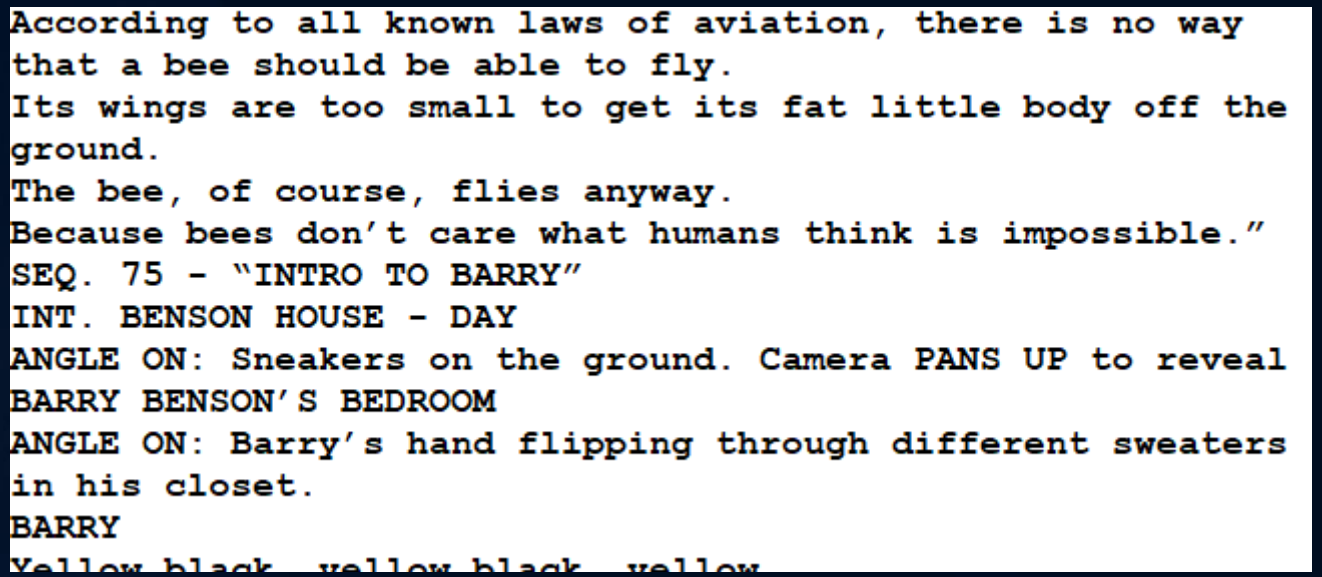
- Why???

```
                mov     dword ptr [ebp+var_18], eax
                mov     dword ptr [ebp+var_18+4], edx
                mov     [ebp+var_1C], 9C40h
                sub     esp, 4
                push    20h             ; n
                push    [ebp+src]       ; src
                push    offset file     ; dest
                call    _memcpy
                add     esp, 10h
                mov     [ebp+i], 0
                jmp     short loc_8048F98
; ---------------------------------------------------------------------------

loc_8048F75:                            ; CODE XREF: stage1+99↓j
                mov     eax, [ebp+i]
                lea     edx, file[eax]
                lea     eax, [ebp+var_F9C]
                push    eax
                push    dword ptr [ebp+var_18+4]
                push    dword ptr [ebp+var_18]
                push    edx
                call    stage2
                add     esp, 10h
                add     [ebp+i], 8
```

Mystery Data

- It turns out there is other data in that block…
- Namely the full script for *Bee Movie*

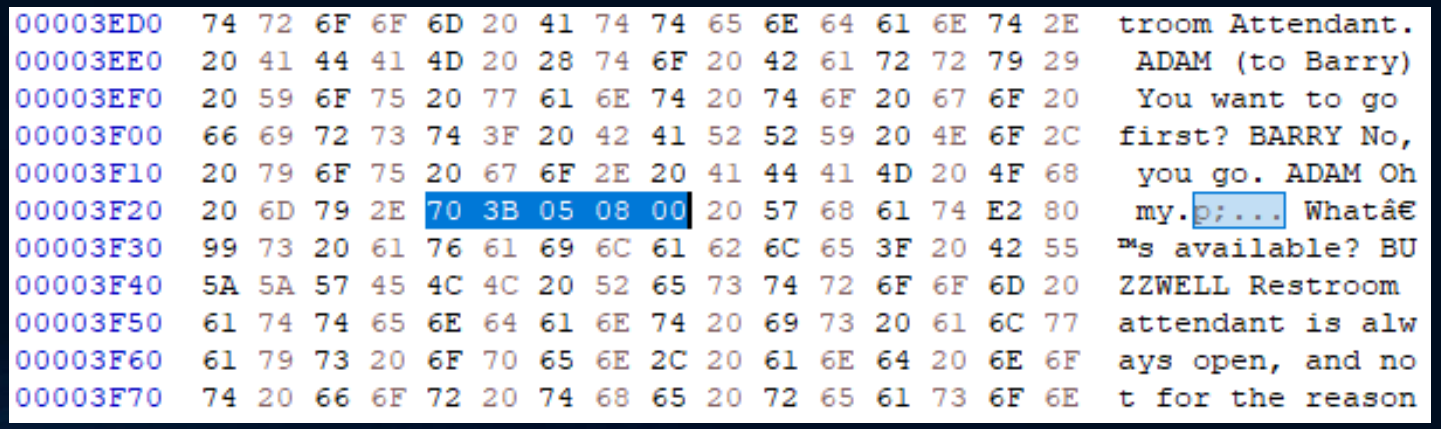


According to all known laws of aviation, there is no way that a bee should be able to fly.
Its wings are too small to get its fat little body off the ground.
The bee, of course, flies anyway.
Because bees don't care what humans think is impossible."
SEQ. 75 - "INTRO TO BARRY"
INT. BENSON HOUSE - DAY
ANGLE ON: Sneakers on the ground. Camera PANS UP to reveal
BARRY BENSON'S BEDROOM
ANGLE ON: Barry's hand flipping through different sweaters in his closet.
BARRY
Yellow black, yellow black, yellow

- What on Earth is this? Another decoy?

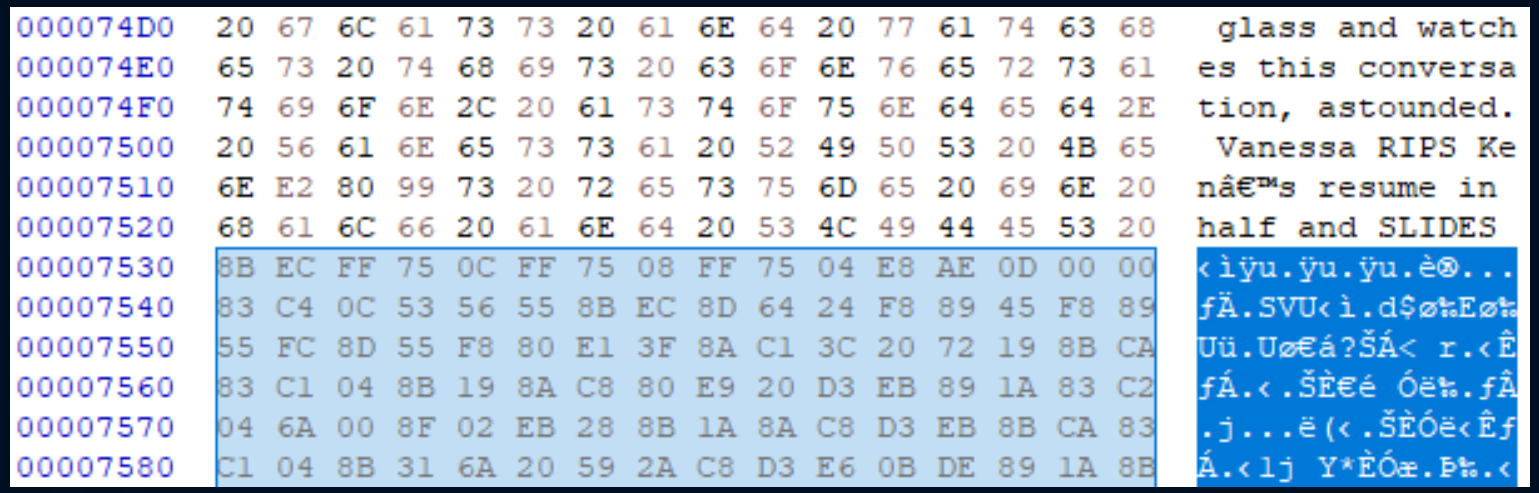
Shellcode

- It turn out this is not *just* the text of the script
- The modified version of `truncate()` functionality is looping over it
- It's set up in such a way that it will trigger a buffer overflow if there is a 0 byte beyond the 16000 byte mark
- And the data contains a 0 at position 0x3F28 (16,168)
- This triggers a jump to address 0x8053B70 (seen just before the 0)



Shellcode (continued)

- That address resolves to 0x8053B70 - 0x804C640 = **0x7530** address inside the script
- When we look there, there is actually binary code there, which is a shellcode for decoding the 3rd part of the flag



```
000074D0  20 67 6C 61 73 73 20 61 6E 64 20 77 61 74 63 68   glass and watch
000074E0  65 73 20 74 68 69 73 20 63 6F 6E 76 65 72 73 61  es this conversa
000074F0  74 69 6F 6E 2C 20 61 73 74 6F 75 6E 64 65 64 2E  tion, astounded.
00007500  20 56 61 6E 65 73 73 61 20 52 49 50 53 20 4B 65   Vanessa RIPS Ke
00007510  6E E2 80 99 73 20 72 65 73 75 6D 65 20 69 6E 20  nâ€™s resume in
00007520  68 61 6C 66 20 61 6E 64 20 53 4C 49 44 45 53 20  half and SLIDES
00007530  8B EC FF 75 0C FF 75 08 FF 75 04 E8 AE 0D 00 00  ‹ìÿu.ÿu.ÿu.è®...
00007540  83 C4 0C 53 56 55 8B EC 8D 64 24 F8 89 45 F8 89  ƒÄ.SVU‹ì.d$ø‰Eø‰
00007550  55 FC 8D 55 F8 80 E1 3F 8A C1 3C 20 72 19 8B CA  Uü.Uø€á?ŠÁ< r.‹Ê
00007560  83 C1 04 8B 19 8A C8 80 E9 20 D3 EB 89 1A 83 C2  ƒÁ.‹.ŠÈ€é Óë‰.ƒÂ
00007570  04 6A 00 8F 02 EB 28 8B 1A 8A C8 D3 EB 8B CA 83  .j...ë(‹.ŠÈÓë‹Êƒ
00007580  C1 04 8B 31 6A 20 59 2A C8 D3 E6 0B DE 89 1A 8B  Á.‹1j Y*ÈÓæ.Þ‰.‹
```

Stage 3: Shellcode Analysis

- The shellcode is sizeable (5Kb)
- At the end of it there are a few hex numbers stored as plaintext strings that are converted to actual numbers:

```
480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
c10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31
d036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb
d1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f
```

- Had to go though the different functions with fine tooth comb and in the end it became clear that most of them are operating on large numbers
  - Essentially a math library - convert, compare, copy, divide with remainder, multiply, etc.
  - Utilizing loops to make such operations possible
- The main function is located at 0xDBE (relative to beginning of shellcode)

Stage 3: Main Function

- After analyzing and removing unimportant operations it essentially boils down to the following steps:
  - Load several big numbers. The relevant ones are:
    - A = 0xc10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31
      (873022861521290123157689568950218112291948903557308140613806839677443481183 21)
    - B = 0xd036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb
      (941778476307813489271457545314274081950503407487335460282572557153120335032 11)
    - C = 0xd1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f
      (948941829825851157522736418698311616042297294876113992679729308339287512740 63)
  - Take 24 bytes of the flag at position 48 - this will be number X
  - Verify that **(X * A) mod C == B**
- If we solve this equation for X we will find the last part of the flag

Stage 3: Solving the Equation

- How do we solve this equation?
- Tried with z3 solver (https://github.com/Z3Prover/z3) but it just spun forever:

```
>>> import z3
>>> x = z3.Int('x')
>>> z3.solve((x * 873022861521290123157689568950218112291948903557308140613806839677443481183 21) % 94894182982
585115752273641869831161604229729487611399267972930833928751274063 == 941778476307813489271457545314274081950 5
034074873354602825725571531203350321 1)
```

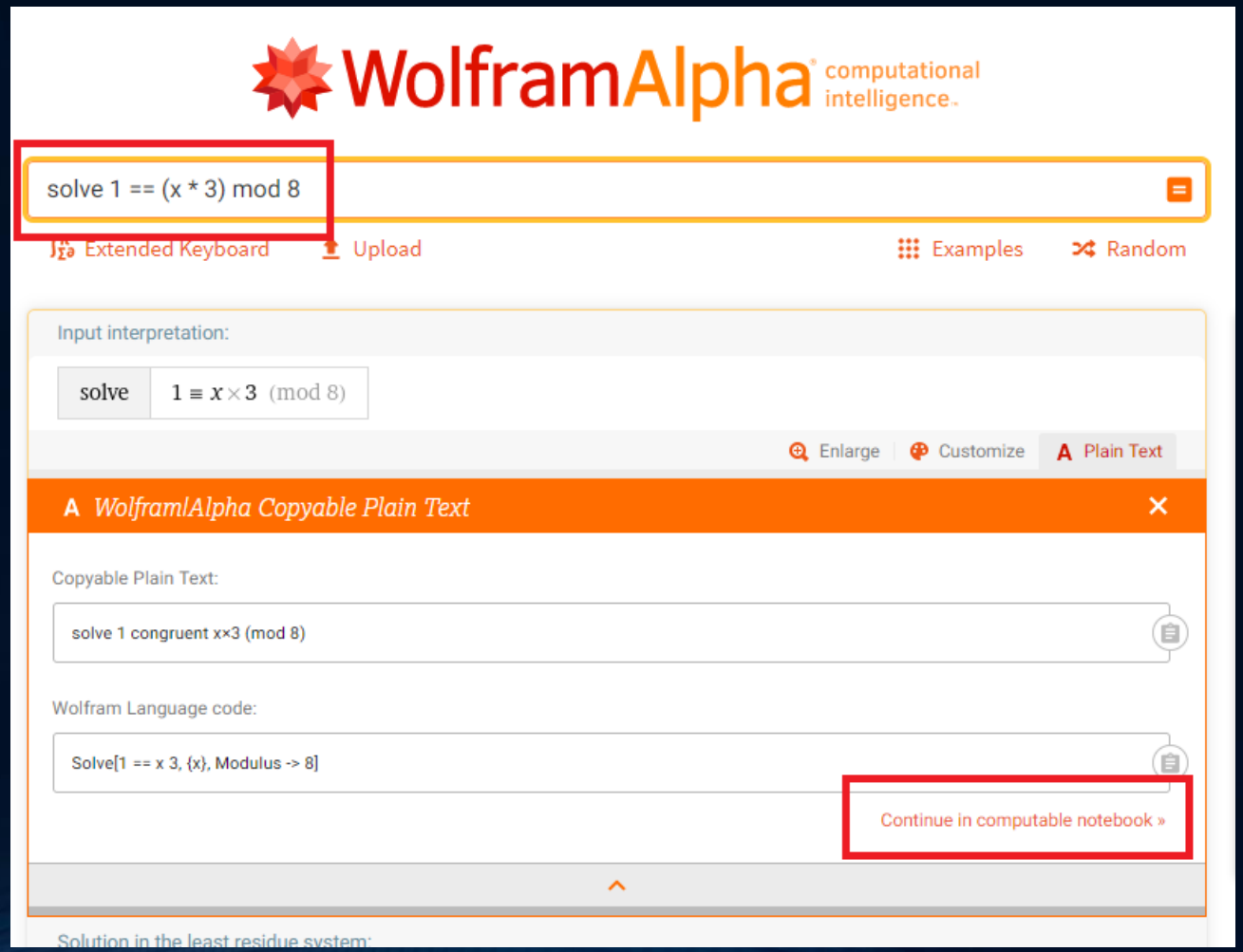
- Luckily, I remembered the following encrypted string seen in the code:

"I hear Wolfram Alpha is good at doing things with big numbers."

- So, let's use Wolfram Alpha to solve it (https://www.wolframalpha.com)

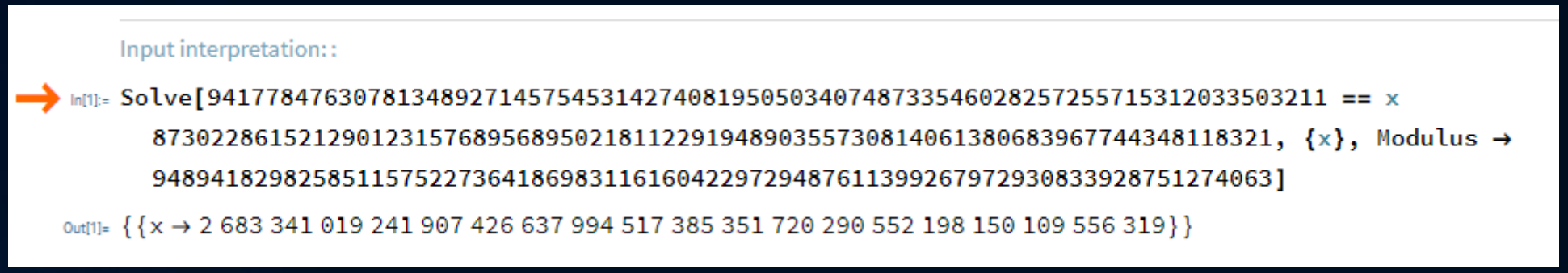
Stage 3: Wolfram Alpha

- Unfortunately the default expression field cuts off long expressions so we must start with a short one:



Stage 3: Solving in the Notebook

- Notebook gives us a solution quickly



- When we grab that number and convert it into characters in Python we finally get the last part of the flag:

```
>>> import binascii
>>> binascii.unhexlify(hex(2683341019241907426637994517385351720290552198150109556319)[2:-1])[::-1]
'_n0_puppi3s@flare-on.com'
```

w3lc0mE_t0_Th3_14nD_0f_De4th_4nd_d3strUct1oN_4nd_n0_puppi3s@flare-on.com

Final Flag Check

- So the full flag is:

w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd_n0_puppi3s@flare-on.com

- After about 30 second delay (depending on the machine) it is confirmed:

```
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one soooper easy

please enter a password friend :) w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd_n0_puppi3s@flare-on.com
hooray! the flag is: w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd_n0_puppi3s@flare-on.com
```

Wrap-up

- 'break' challenge was the "worst case" combination of various RE avoidance techniques:
  - ptrace and signals used for execution
  - function overrides
  - tons of decoys
  - custom encryption
  - complex calculations
- …which is what made it so satisfying to learn and solve ☺
- If you are interested in RE, Flare-On is a great competition to try
- You will learn a lot of practical techniques and get to reverse pretty much any type of code imaginable

Questions?

Thank you!