Kernel Sanders: CSAW ESC'19 Final Report

Grant Hernandez*, Hunter Searle*, Owen Flannagan*, Claire Seiler*, Kevin R.B. Butler*
*University of Florida, Gainesville, FL, USA
{grant.hernandez, huntersearle, owenflannagan, cseiler, butler}@ufl.edu

I. Introduction

First, to enable collaboration amongst team members, we set up a GHIDRA shared project on our own server to share reversing progress. All relevant challenge binaries were ARM-32 Cortex-M4 (TeensyChallengeSetX.ino.elf). These contained all of the `challenge_X` functions that we had to reverse. The AVR binaries were used for blinking the LEDs, polling the buttons, and displaying to the LCD. It also acted as the I^2C bus master, with the Teensy acting as a slave with address `0x01`. The single most difficult part of reversing these challenges with GHIDRA was the broken RFID stack variables. GHIDRA was unable, for nearly all challenges, to make the offsets into the RFID data easily apparent.

To assist us with the problem of stack addresses, we used ANGR, a python framework for analyzing binaries. In order to analyze binaries, ANGR lifts the file into Valgrind's VEX intermediate representation (IR), then uses both static and dynamic ("concolic") analysis. We used ANGR to hook all memory reads and writes. If the R/W fell in the range of the RFID data on the stack, we printed `CARD READ: XX` where XX was the hex offset into the card data. This alone saved us from manually counting offsets across stack frames. Beyond finding the address of card reads, ANGR allowed us to automatically solve many of the simpler challenges. More details are shown in Section II-A.

The highlights of our report are:

- We employed concolic analysis using ANGR to avoid reversing as many challenges as possible
- We achieved arbitrary code execution on challenge D-bounce (Section V-A)

All hashes for our solved challenges are in the Appendix under Section A. We were able to solve 16/18 challenges for a total of 1810 points. Our video demo is available at https://drive.google.com/open?id=1Dxu0LSNhNxHRTTTYGJKsosiagBaUCiCX

II. Challenge Set A

A. Lounge

Lounge was, at first glance, a difficult challenge due to all of the emulated floating point instructions. Further reversing revealed that only two bytes of card data are used to determine the win condition of `a * b == 0x18af`. This means the keyspace is 2^{16} – an easily bruteforceable amount. To enable us to bruteforce without manually reflashing card data, we used ANGR for dynamic analysis. To start, we created an ANGR project:

```
import angr
proj = angr.Project("A/TeensyChallengeSetA.ino.elf")
```

Then we created a `blank_state`, disabled symbolic memory, set the starting PC to `challenge_0`, and explored until the goodboy or end of the function:

```
st = proj.factory.blank_state()
st.regs.pc = proj.loader.main_object.symbols_by_name["
    _Z11challenge_06packet"].linked_addr
st.options |= set(["ZERO_FILL_UNCONSTRAINED_MEMORY"])
mgr = self.proj.factory.simgr(st)
mgr.use_technique(angr.exploration_techniques.Explorer(
    find=[0xc21], avoid=[0xc51]))
mgr.run()
```

This performs purely concrete execution until an address in the `find` or `avoid` sets is found. In this case, because the RFID data was assumed to be zero, the SimulationManager ends with one state in the "avoid" stash. This run took exactly 30 seconds, which is quite a slowdown compared to a real execution environment. This is because ANGR interprets VEX Intermediate Representation (IR) instead of native machine code, in addition to performing expensive memory and register bookkeeping. This can incur slowdowns of 100 - 1000x, depending on the instructions being emulated. To alievate this slowdown, ANGR provides addtional execution engines, such as Unicorn, which executes native instructions, to burn through concrete instruction traces. Unfortunately, ANGR's version of Unicorn *does not* support ARM, preventing this speedup.

With these constraints, it looked as if concrete bruteforcing with ANGR would be too expensive. Ironically, switching to symbolic execution let us discover more than one solution to this problem in less than two hours of wall-clock time. Switching to symbolic execution involved investigating which offsets into the card data were being read by the challenge function. To do this, we hooked all memory reads during execution and printed when a read address fell in the range of the RFID card data on the local stack frame:

```
# Determined by breakpointing in angr and correlating to
    the output of
# 'debugPrintPacket'
WHITE_CARD_START_ADDR = 0x7fff0000-0xf
WHITE_CARD_SZ = 16*64
WHITE_CARD_END_ADDR = WHITE_CARD_START_ADDR +
    WHITE_CARD_SZ
BUTTON_OFFSET = WHITE_CARD_START_ADDR + WHITE_CARD_SZ +
    48
```

```
def print_card_offsets(state):
    expr = state.inspect.mem_read_address
    # the address could be symbolic, so get 'a' solution
    expr_val = state.solver.eval(expr)

    if expr_val >= WHITE_CARD_START_ADDR and expr_val <=
         WHITE_CARD_END_ADDR:
        offset = expr_val - WHITE_CARD_START_ADDR
        print("CARD READ: %x (%s)" % (offset, str(expr))
            )
    elif expr_val == BUTTON_OFFSET:
        print("!!!!!! BUTTON READ !!!!!!")

st.inspect.b('mem_read', when=angr.BP_AFTER, action=
    print_card_offsets)
```

The `WHITE_CARD_START_ADDR` was determined by manual inspection by stepping through execution with ANGR. We enable this breakpoint on every challenge we solve going forward. In this case, the card offsets were `0x4c` and `0x4d`. Once we had determined these, we were able to set these offsets as symbolic variables:

```
st.memory.store(WHITE_CARD_START_ADDR+0x4c, st.solver.
    BVS("input1", 8))
st.memory.store(WHITE_CARD_START_ADDR+0x4d, st.solver.
    BVS("input2", 8))
```

These are the only variables in memory that we made symbolic (the `ZERO_FILL_UNCONSTRAINED_MEMORY` ensures this). We also track when the button values are read by a challenge function. This offset was determined by looking at the static RFID structure in GHIDRA. Next, to speed up the execution process, we added lightweight parallelism. We executed until we received a found state with the Explorer PathTechnique shown earlier:

```
from multiprocessing import Pool, cpu_count

...

def exec_once_lounge(self, state):
    """ Executed in another process """
    mgr = self.proj.factory.simgr(state)
    mgr.run(n=20)
    return [mgr.active, mgr.found]

def join_results(omgr):
    mgr.active += omgr[0]
    mgr.found += omgr[1]

# get some initial paths
mgr.run(n=4)

pool = Pool(processes=cpu_count())

while not mgr.found:
    print(mgr)

    if len(mgr.active) == 0:
        time.sleep(1)
        continue

    active_st = mgr.active.copy()
    mgr.drop(stash='active')

    print("Distributing %d states" % len(active_st))

    for a in active_st:
        pool.apply_async(exec_once_lounge, args=(a,),
            callback=join_results)
```

Running the above code on dual Intel Xeon CPU E5-2630 v4 @ 2.20GHz CPUS with 40 cores total, we were able to find two paths, at which point the execution halted. To help pretty-print the card data table and buttons, we designed a helper that evaluates the symbolic or concrete card data from an execution state:

```
def print_table(self, state):
    table = state.solver.eval(state.memory.load(
        WHITE_CARD_START_ADDR, WHITE_CARD_SZ), cast_to=
        bytes)
    buttons = state.solver.eval(state.memory.load(
        BUTTON_OFFSET, 1), cast_to=int)

    arr = []
    for i in range(64):
        arr += [[c for c in table[i*16:(i+1)*16]]]

    output = []
    output += ["#     0  1  2  3  4  5  6  7  8  9  a  b
         c  d  e  f"]
    output += ["p = ["]

    for i, row in enumerate(arr):
        eol = "," if i < 63 else "]"
        row = ", ".join([(("0x%02x" % x) if x != 0 else
            "0") for x in row])
        output += ["     [" + str(row) + ("]%s # %x" % (
            eol, i))]


    output += ["a = 0x%x" % ((buttons >> 4) & 0xf)]
    output += ["b = 0x%x" % (buttons & 0xf)]

    print("\n".join(output))
```

Calling `print_table` allows us to create `sender.py` files by just copying and pasting the result. We also have a mode to directly program a card if ANGR is run on the local machine.

As we did not need to reverse engineer this challenge at all, except to find the goodboy and badboy basic block addresses (0xc21 and 0xc51), no discussion is necessary and if this kind of "lock" was used in the real world, it would quite ineffective as the key is too small. The two solutions we found for this challenge are in Section A.

B. Closet

With the basic ANGR framework created for the previous challenge, we were able to easily support a new challenge. All that needed to be changed was the initial starting function address. Unlike the previous challenge, symbolic execution with ANGR did not fare so well. We encountered constraint explosion due to a symbolic memory read on line 18 below:

```
1  char table[128];
2  char key[12] = "ESC19-rocks!"
3  bool good = true;
4
5  for (int i = 0x5c; i < 0x84; i++) {
6    if (i < 0x70) {
7      // stored in table at +0x6c
8      table[i + 0x10] = RFID[i];
9      Print::println((Print *)&Serial,i + -0x5c);
10   } else if (0x7f < i) {
11     table[i] = RFID[i];
12     Print::println((Print *)&Serial, i + -0x6c);
13   }
14 }
15
16 for (int i = 0; i < 12; i++) {
17   // this causes angr to blow up as it is a symbolic
         index
18   if (key[i] != table[(uint)table[i + 0x6c] + 0x6c])
19     good = false;
20 }
21 ...
```

Assuming the first `table` load was symbolic, then the next table load's address would be symbolic. ANGR instead of loading from a single address, loads from 256 addresses within the table *simultaneously*. This causes the result of the last table lookup to be the disjunction of 256 separate memory loads. These yield massive constraints which get passed on to the Z3constraint solver, which greatly slows down. The time to determine the satisfiability increases each time through the loop. This ends up taking so long that 99% of the time executing is spent in the solver. We tried to solve this by preconstraining our symbolic card input to resonable values, but this still causes slow downs and final card data outputs to be wrong. Instead for this challenge, we manually solved it by dumping the concrete stack data in ANGR and measuring the offset from the table load to the already in-memory key `ESC19-rocks!`. This offset was `0x18+i`. The two lines needed to solve this concretely with ANGR are below:

```
for i in range(0xc):
    st.memory.store(WHITE_CARD_START_ADDR+0x5c+i, pack("<
        b", 0x18+i))
```

From a security perspective, it should be noted that we are able to read outside the bounds of the `table` variable. In this case, it was relevant as the key was outside of the table (we did not need to pass it in via the RFID table).

C. Cafe

The cafe challenge involved a linear transformation of a template challenge hash with multiple XORs and various logic. Like previous challenges, we worked smart and avoided any reversing and just threw ANGR at the challenge. The catch for this challenge versus others is that there is no dependent branch that indicates whether the challenge was solved or not. Instead the template challenge hash is transformed in various ways and MUST equal the string `solved challenge cafe abcdefg`.

We solved this challenge using ANGR in symbolic mode. At this point, we wrapped all our ANGR usage in a class with helpers to aid the development. For more information read the `angresc.py` file included with the challenge submission. The relevant lines from the solver are included below:

```
self._set_start_symbol("_Z11challenge_26packet")
addr = self.sym.linked_addr

self._hook_prints()
st = self._get_start_state(addr, ['
    SYMBOL_FILL_UNCONSTRAINED_MEMORY'])

mgr = self.proj.factory.simgr(st)
# We used Veritesting to aggressively merge states and
    save execution time
# Without this, execution took much more time
mgr.use_technique(angr.exploration_techniques.
    Veritesting())

mgr.run()

# The final state becomes unconstrained with it returns
    from the challenge
# function as the saved LR is left as symbolic (
    intentionally)
if not mgr.unconstrained:
    print("Analysis failed")
    return

s = mgr.unconstrained[0]

fixed = 'solved challenge cafe abcdefg'

# address of challResult seen in GHIDRA
challResult = 0x1fffa140

# Constrain the hash
for i in range(len(fixed)):
    s.solver.add(s.memory.load(challResult+i, 1) == ord(
        fixed[i]))

# Eval the string at the address given the constraints
strout = self.read_string(s, challResult)
print("ChallResult: " +  repr(strout))

self.print_table(s)
```

The highlights of this challenge are the use of Veritesting [1] and additional constraints to get the desired output. Veritesting enables aggressive state merging, shoving more responsibility to the solver. At the end of the function only a single symbolic state with all the possible symbolic constraints for each byte in the challenge hash OR'd with each out. This state reaches the end of the function and becomes unconstrained as its saved LR is symbolic to prevent returns from the challenge function. Then using the single unconstrained path, we add additional constraints to the challenge hash variable gleaned from GHIDRA to yield the correct card table upon printing.

D. Stairs

We were given the solution for stairs, but we solved it with ANGR anyways.

```
self._set_start_symbol("_Z11challenge_36packet")
addr = self.sym.linked_addr

self._hook_prints()
st = self._get_start_state(addr, ['
    SYMBOL_FILL_UNCONSTRAINED_MEMORY'])

mgr = self.proj.factory.simgr(st)
mgr.use_technique(angr.exploration_techniques.Explorer(
    find=[0xf61], avoid=[0xf85,0xf39]))

mgr.run()

s = mgr.found[0]
mgr_final = self.proj.factory.simgr(s)
mgr_final.run()

self.print_table(s)
```

No issues were experienced with solving this with ANGR.

III. Challenge Set B

A. Mobile

This challenge involved a simple algorithm that, based on the values of an array through which it iterated, would select characters from a lookup table (LUT) containing the ASCII alphabet. The most important code is the following:

```
lut_it = 0;
hash_it = 7;
for(int i = 1; i < 0x1e; i++){
```

[1] https://github.com/angr/angr/blob/master/angr/analyses/veritesting.py

```
  if (done == 0) {
    if (indexes[i] == indexes[i - 1]) {
      lut_it = lut_it + 1;
    }
    else {
      if ((indexes[i] == 0) || (indexes[i - 1] == 0)) {
        if (indexes[i - 1] != 0) {
          challHashGen[hash_it] = LUT[lut_it + indexes[i
              - 1] * 3];
          hash_it = hash_it + 1;
        }
        lut_it = 0;
      }
      else {
        done = 1;
      }
    }
  }
}
```

In order to select the nth character from the LUT, `lut_it` should be $n\%3$ and indexes[i-1] should be $n/3$. To achieve this, we set an array with $n\%3+1$ instances of the value $n/3$, followed by a single zero. This pattern was repeated for each character that was needed to achieve the correct output message. ANGR was attempted to be used for this, but due to the symbolic read similar to A-closet, we opted for a manual reverse engineering approach.

B. Dance

Previous challenges had been self-contained, but Dance employed a library to perform Blake-256 hashes. It should be noted that ANGR is unable to solve any challenge involving cryptographically *strong* hash functions. This is because symbolically executing a hash function would pass unsolvable constraints to the underlying constraint solving engine. If the engine *was* able to deduce a solution, this would constitute a break of the hash function.

To solve the challenge without ANGR, the correct 8-character password needed to be passed in. The function performs a hash of this input and compares it against a fixed digest. To determine if this hash was already cracked, we reconstituted it from the decompiled GHIDRA output: `5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8`. Google'ing this hash lead to many hash cracking websites that showed this was infact a SHA-256 hash of the word "password". Placing this at the starting offset of 0x93 led to the solve.

C. Code

Like the previous challenge, Code involved a hash function `H45H`. Examining the functions for constants and digest size indicated that this was MD5. The input to MD5 was the string "imjustrandomdatathathasnomeaningwhatsoever!" and a single character from the RFID card at offset 0x9b. The digest was compared to a fixed hexstring. A quick python brute forcer was written to find the correct byte:

```
import struct
import hashlib

target = "242b461d0b97cca55e5d62372b770ab4"

assert len(target) == 32

for i in range(256):
    inp = "imjustrandomdatathathasnomeaningwhatsoever!"
        + struct.pack("B", i)
    res = hashlib.md5(inp).hexdigest()
    assert len(res) == len(target)
    if res == target:
        print(inp, i)
        break
```

The correct byte was 'L'. This led to the solve. There was a base64 string as a hint, but all it said was "berger king". It's unclear how this was a hint.

D. Blue

We were unable to solve this challenge. We tried using ocl-hashcat with many rules/wordlists to get the 8-character password, but to no avail.

IV. Challenge Set C

A. Uno

To start, the hint for this challenge was "Is OISC 1337?". Looking up OISC lead to the One Instruction Set Computer wikipedia article. From previous experience, we know that any arbitrary computation could be created with a `subleq` instruction. This instruction performs a subtraction of two memory operands and branches if their result is less than or equal to zero. Reading, retyping, and renaming in GHIDRA confirmed that this was the instruction set being used:

```
int MEM[0x5f];
// each char +3 from "solved ..."
// need to -3 to restore
char * TABLE = "vroyhg#fkdoohqjh#xqr#defghijklm";

// initialize memory
for (int i = 0; i < 0x5f; i++) {
  if (i < 0x30) MEM[i] = (int)RFID[0x200+i];
  else if (i < 0x40) MEM[i] = (int)RFID[0x240+i];
  else MEM[i] = (int)TABLE[i];
}

while ((hashAddr = hashAddrCpy, PREV_PC = PC, -1 < PC &&
     (PC < 0x3d))) {
  PC_1 = PC + 1;
  PC = PC_1;
  CMP1 = MEM[PREV_PC];
  PC = PREV_PC + 2;
  CMP2 = MEM[PC_1];
  PC = PREV_PC + 3;
  BRANCH_PC = MEM[PREV_PC + 2];
  if (CMP1 == -1) break;
  if ((CMP2 == -1) && (hashAddrCpy < 0x1f)) {
    hashAddrCpy = hashAddrCpy + 1;
    challHash[hashAddr] = (char)MEM[CMP1];
  }
  else {
    // SUBLEQ OISC
    MEM[CMP2] = MEM[CMP2] - MEM[CMP1];
    if (MEM[CMP2] < 1) {
      PC = BRANCH_PC;
    }
  }
}
```

The trick for this challenge was that the challHash was uninititialized and had to be assigned to by the OISC loop. Luckily, a transformed version of the output was placed into the MEM region at offset 0x40. This was loaded from a fixed table and was just the "solved" string with a

character shift of +3. Good thing we have an instruction dedicated to subtracting!

Next began the task of writing a `subleq` program to shift and store the string. To aid development, we created a two-pass assembler (available under `solutions/C-uno-8/asm.py`). Our resulting subleq assembly was the following:

```
; A, B, C are integers and are treated as addresses
; MEM[B] = MEM[B] - MEM[A]
; if (MEM[B] < 1)
;   PC = C

; for(int i = 0; i < 31; i++) {
;   MEM[0x40+i] = MEM[0x40+i] - 3;
;   output(MEM[0x40]);
; }
;

loop:
    subleq 0x40, pos3, -1 ; table[i] -= 3
loop2:
    subleq -1, 0x40, -1   ; output(MEM[i])
    subleq 1, neg1, -1    ; self-modifying code
    subleq 3, neg1, -1    ; self-modifying code
    subleq TA, TA, next   ; TA = 0
next:
    subleq TA, tablemax, next2  ; TA = -tablemax
next2:
    subleq TB, TA, next3  ; TB = -TA
next3:
    subleq TB, loop, end  ; tablemax - MEM[0]
    subleq TB, TB, loop   ; goto loop
end:
    subleq -1, -1, -1     ; exit program

; Data region
pos3:
  dd 3
neg1:
  dd -1
tablemax:
  dd 0x5f
TA:
  dd 0
TB:
  dd 0
```

Compiling this yielded [30, 64, -1, 64, -1, -1, 31, 1, -1, 31, 3, -1, 33, 33, 15, 32, 33, 18, 33, 34, 21, 0, 34, 27, 34, 34, 0, -1, -1, -1, 3, -1, 95, 0, 0] for a total length of 35 words. This was written to card offset 0x200 as bytes and the challenge was solved. ANGR was used to test and debug the concrete solution without reflashing the card for each iteration.

B. Game

Our first clue in solving this challenge was a function called `findBestMove`. This suggested to us that the solution would require putting game moves onto the card in order to play against the program. Within the `findBestMove` function is another function called `minimax`, which is a common algorithm for finding optimimal moves in simple games. An examination of `minimax` made it clear that the game being played is tic-tac-toe. The original board state is saved in a variable in the challenge function. The game begins with the program having moved twice (player X), and the keycard (player O) having moved once:

```
xx_
_o_
___
```

Therefore, the keycard moves first after being scanned. It was a simple matter to plan out our moves to ensure that the keycard ties with the program. The sequence of moves were read in starting at offset 0x9c. The move encoding was each byte was a move with the top nibble being the row and the bottom nibble being the column. The moves to tie were [r, c] (0, 2), (1, 0), and (2, 2). This left the board in the state of a tie, leading to the win condition.

C. Break

This challenge was aptly named. ANGR chewed through it.

```
st = self._get_start_state(addr, ['
    SYMBOL_FILL_UNCONSTRAINED_MEMORY'])

mgr = self.proj.factory.simgr(st)
# explore to the goodboy
mgr.explore(find=[0x11b9])

self.print_table(mgr.found[0])
```

The offsets 0x9f and 0xa0 were set to one and the buttons set to a = 0x4, b = 0x6.

D. Recess

This challenge was *also* aptly named.

```
st = self._get_start_state(addr, ['
    SYMBOL_FILL_UNCONSTRAINED_MEMORY'])âŃĚ
mgr = self.proj.factory.simgr(st)

# explore to the goodboy
mgr.explore(find=[0x1291])

self.print_table(mgr.found[0])
```

The offsets 0xa1 - 0xa4 were set to the string "g00d" to get the solve.

V. Challenge Set D

A. Bounce

This challenge was short but different than all of the others. The challenge hash not being filled in the challenge function itself. Further investigation revealed that the `fillChallengeHash` called only during `setup` would fill and send the hash. This would only happen if the `use` global boolean was set to true. This boolean was set to true when conditions were met in the actual challenge function.

At first glance, it would seem that calling this function again is impossible, but knowing how return addresses on ARM are saved, we notice that if the right card data is provided, we can overflow the stack with arbitrary data. Using ANGR for dynamic analysis, we solved the required input constraints to make `use = true`. This included control over the saved LR on the stack. Here is the solver script:

```
mgr = self.proj.factory.simgr(st)

mgr.use_technique(angr.exploration_techniques.Explorer(
    find=[0x876, 0x877], avoid=[0x874,0x875]))
st.memory.store(WHITE_CARD_START_ADDR, b"\x00"*
    WHITE_CARD_SZ)
```

```
# Returning right towards the fillChallengeHash function
target_pc = self.obj.symbols_by_name["
    _Z17fillChallengeHashv"].linked_addr

print("[+] Exploit target PC %08x" % target_pc)

stage1 = b"\x00"*12 + pack("<I", 12) + b"\x00\x00\x00\
    x00" + pack("<I", target_pc)

# clear the white card
st.memory.store(WHITE_CARD_START_ADDR, b"\x00"*
    WHITE_CARD_SZ)

# for each bit that is set, read a byte from the payload
     (24 bytes)
st.memory.store(WHITE_CARD_START_ADDR+0x100, b"\xff"*3)
st.memory.store(WHITE_CARD_START_ADDR+0xc0, stage1)

# Keep buttons symbolic
st.memory.store(BUTTON_OFFSET, st.solver.BVS('button',
    8))
```

Setting the buttons to a = 0x1, b = 0xd allowed the challenge to be solved.

a) Arbitrary Code Execution: Given that we are able to fully control the instruction pointer, we can redirect it to a controlled space in memory to execute ARM Thumb-2 shellcode. The Cortex-M4 does not have any mitigations (ASLR, XN, MMU, etc.), making this trivial. We jump into the global RFID variable at [0x1fff976d + 0x110] (we just change `target_pc` in the ANGR script) to start executing the following code:

```
.section .text
.align 2
.syntax unified

adr r7, putchar
ldrh r7, [r7]

adr r6, hacked

looper:
  ldrb r0, [r6]
  blx r7
  ldrb r0, [r6, #1]
  blx r7

b looper

; usb_serial_putchar function
putchar:
.word 0x3dad

hacked:
.ascii "HACKED\n"
```

This will print "HA" over and over until the watchdog timer resets. We ran into many issues getting more code to execute as we believe it was being cut off during the RFID reading process. Hence, HA instead of HACKED. See the Bounce challenge directory for the Makefile.

VI. Challenge Set E

A. Steel

This challenge involved MD5 hashing like B-code, with the catch that a hash was performed multiple times to increase "security". The input into the hash function is a single byte that is determined by some transformations on card data (easily solved by ANGR). Therefore, we must first determine the single byte of input into the MD5 hashing rounds to solve this problem. At first glance, this problem seems like it can be solved with a trivial python bruteforcer:

```
import hashlib

target = "703224f765d313ee4ed0fadcf9d63a5e"

for i in range(256):
    obj = hashlib.md5()
    obj.update(chr(i))
    res = obj.hexdigest()

    for i in range(9):
        obj.update(res)
        res = obj.hexdigest()

    if res == target:
        print("FOUND: " + chr(i))
        break
```

This proved to be wrong due to the implmentation details of padding during the calls to `H45H::Final`, which Python's hashlib did not respect. To account for this, we downloaded the library that H45H was compiled from: Hashlib++.[2] We wrote the following program to mirror what we saw in GHIDRA:

```
unsigned char buff[16] = "";|âŃĚ
std::string target = "703224f765d313ee4ed0fadcf9d63a5e";

for (int i = 0; i < 255; i++) {
  MD5 * md5 = new MD5();
  HL_MD5_CTX ctx;
  unsigned char inp = i;

  memset(&ctx, 0, sizeof(ctx));

  md5->MD5Init(&ctx);
  md5->MD5Update(&ctx, &inp, (unsigned int)1);
  md5->MD5Final((unsigned char *)buff, &ctx);

  std::string hexdigest = convToString(buff);

  for (int j = 0; j < 9; j++) {
    md5->MD5Update(&ctx, (unsigned char *)hexdigest.
        c_str(), 32);
    md5->MD5Final((unsigned char *)buff, &ctx);

    hexdigest = convToString(buff);
  }

  if (hexdigest == target) {
    std::cout << "Got it: " << inp << std::endl;
    break;
  }

  delete md5;
}
```

Compiling and running with `g++ -I build/include/ -o test test.cpp build/lib/libhl++.a && ./test` still did not find any results. Debugging with ANGR to compare the output of the second Final call showed a mismatch. Digging into the Final function source code yielded the answer:

```
1  void MD5::MD5Final (unsigned char digest[16],
       HL_MD5_CTX *context)
2  {
3     ...
4
5     /*
6      * Zeroize sensitive information.
7      */
8     MD5_memset ((POINTER)context, 0, sizeof (*context))
          ;
9  }
```

[2] http://hashlib2plus.sourceforge.net/

The memset on line 8 was NOT in the compiled version running on the Teensy. Commenting this line out allowed the test program to find the correct hash input of semicolon (';'). With this initial input, we could now use ANGR to solve the first transforms with the known ending constraint of semicolon:

```
st.memory.store(WHITE_CARD_START_ADDR+0x191, st.solver.
    BVS('input', 8*3))

mgr = self.proj.factory.simgr(st)
mgr.explore(find=[0x1796+1])

s = mgr.found[0]
s.solver.add(s.memory.load(s.regs.r7+0x8c, 1) == ord(';'
    ))

self.print_table(s)
```

B. Caeser

We were unable to solve this challenge due to time constraints.

C. Spiral

This challenge was easily solved by ANGR and did not require any reversing:

```
st = self._get_start_state(addr, ['
    ZERO_FILL_UNCONSTRAINED_MEMORY'])
mgr = self.proj.factory.simgr(st)

st.memory.store(WHITE_CARD_START_ADDR+0x18d, st.solver.
    BVS("input", 8*4))
mgr.explore(find=[0x1e05], avoid=[0x1e2b])

s = mgr.found[0]
self.print_table(s)
```

D. Tower

Examining the challenge showed it was comparing against a SHA-256 hash again, but this time with an input length of 13. Even if the password was only lowercase letters, this would require more than $2x10^{18}$ hashes – far exceeding a bruteforceable limit. There was a base64 encoded string above the hashing that decoded to 'ht'. We assumed this stood for hash table and wasted time looking for one. We also used ocl-hashcat with as many wordlists and rule sets as we could given the time, but no matches were found. Further investigation showed that there were more base64 strings encoded throughout. We collected and decoded them all below:

```
parts = ['ht', 'tps', 'geW', '://pas', '.com/', 'Ve', '
    in', 'teb', 'mJP']
```

We rearrainging them into https://pastebin.com/VegeWmJP which led to the password `ndixlelxivnwl`! We burned this on to the card starting at offset 0x180 and got the solve.

VII. Challenge Set F

A. Spire

The final challenge looked resonably straight forward to have ANGR solved. But when running the resulting table on the device, it reset. Examining the code further showed a strange variable assignment to `_Reset`. Looking at the assembly confirmed that there was a store instruction that was writing by default to the reset vectors, causing a processor fault. In order for our table to process and to see any debugging information we need to avoid this reset. To do this, we employ the overflow given to us when offset 0x36f is non-zero. This allows us to write an arbitrary amount of data onto the stack, overflowing into nearby variables. One of these variables is the store address that was causing the reset. By replacing this address with a known global address that is writable we can avoid the reset condition. Additionally, past this variable is a final comparision required to be non-zero to solve, which we can also overwrite. The concrete solution in ANGR is below:

```
st = self._get_start_state(addr, ['
    ZERO_FILL_UNCONSTRAINED_MEMORY'])âŃĒ

# the amount of bytes to overwrite on the stack (
    negative)
st.memory.store(WHITE_CARD_START_ADDR+0x280, pack("<I",
    0) + pack("<i", -(4*3)))
# needs to be non-zero inorder to allow for overwrite
st.memory.store(WHITE_CARD_START_ADDR+0x36f, pack("B",
    1))
# must be non-zero to print debugging information
st.memory.store(WHITE_CARD_START_ADDR+0x340, pack("B",
    1))
# The region which is written on to key stack variables
# 0x2c3 must be 1 to pass the final check, 0x2c7 a dont'
    care,
# 0x2cb MUST be a valid writable memory address to avoid
     resets
st.memory.store(WHITE_CARD_START_ADDR+0x2c3, pack("<IIIB
    ", 1, 0, 0x1fffa140, 0))
```

ANGR was instrumental in showing which addresses and which offsets of the card data were written. Like challenge D-bounce, this challenge can be exploited to achieve arbitrary code execution as the saved LR can be overwritten.

Appendix

Challenge Hashes

```
Challenge A
-------------------
0 - lounge: 643a6fa20b171fdf3a9e7e1975ce62892fde9cecf2056a73d85fa2d0802d3000 (100 pts)
          1d98b315de9dc5a80ee260e571e5358d07a5146248353ed10702176960859f71 (alternative)
1 - closet: 293f7b60b994512db99836ae7d5bab88b2d0089f90fcf6d51b95b374200dc20f (100 pts)
2 - cafe:   98bc5b1a13fdda3cca488c06ddac0aa5c5449c8a9294a9e5c297806e7faff007 (130 pts)
3 - stairs: 396f4b1cdf1cc2e7680f2a8716a18c887cd489e12232e75b6810e9d5e91426c7 (50 pts)

Challenge B
-------------------
4 - mobile: 68514b8771e5894c799f540855afbc36ef70db34d274a64a8d4271bc1f188379 (100 pts)
5 - dance:  e631b32e3e493c51e5c2b22d1486d401c76ac83e3910566924bcc51b2157c837 (130 pts)
6 - code:   372ded6746e45ef7c8ad5a22c5738a4b5aa982da66bc8a426aa1cca830d05af3 (50 pts)
7 - blue:   [UNSOLVED]

Challenge C
-------------------
8  - uno:    4842370a583df2fd6328d4a30b09c6bb58d0df767691872b7263e01aed9651cd (200 pts)
9  - game:   63c0b41f89bbf493ba791c092b3e5473e243b9c16666f1e5eaa82bc52eeb1613 (150 pts)
10 - break:  ae4be3d07679f53cf7fd0ed9669d06bc3b22c5554c81e3bad04986bb7ab91db1 (70 pts)
11 - recess: 370815b8d8fde829f5c35f893d0b4139d61a775baa4181fcac1fffe014bde9ea (100 pts)

Challenge D
-------------------
8 - bounce:  bb8d9065d2656d3ab62ab650b0543fe73844f4df52f5f5d60cc10b31ae6086ac (150 pts)

Challenge E
-------------------
12 - steel:  102a3ac6eb0ef2cd010d4ef1776571cdc2af0727ffd8c839295e252ab4a25211 (100 pts)
13 - caesar: [UNSOLVED]
14 - spiral: 26ee8470c732dfc821bbe0561b446dc8086560e4e222b22e6a74e559d90a7d61 (130 pts)
15 - tower:  b019c48299dd33ec6fdc94da9d5ad06018549ee58f4a829a44d15e6980c22cbb (100 pts)

Challenge F
-------------------
17 - spire: 01624bc0041de97a87defd12414c91c513de8fd1d41c4a139438be1965b187c3 (150 pts)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ TOTAL POINTS: 1810 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```