NUSec: CSAW Embedded Security Challenge Report 2020

Dennis Giese (Lead)
NUSec
Northeastern University
Boston, MA

Cameron Kennedy
NUSec
Northeastern University
Boston, MA

Erik Uhlmann
NUSec
Northeastern University
Boston, MA

Guevara Noubir, PhD. (Advisor)
NUSec
Northeastern University
Boston, MA

***Abstract*—Internet-of-Things devices have become ubiquitous in modern-day smart infrastructure. With tens of billions of IoT devices estimated to be connected today, the security of IoT is critical, but many devices still suffer from security vulnerabilities. The challenges provided in this competition involved an IoT Wi-Fi access point running on a RISC-V single-board computer. This report summarizes and discusses how we (NUSec) reverse-engineered and analyzed the provided firmware binaries to solve the provided challenges.**

***Index Terms*—CSAW-ESC, RISC-V, Ghidra, Static Analysis, Reverse Engineering, Theorem Proving**

I. Introduction

The NUSec CTF team leveraged a number of static and dynamic analysis techniques in order to solve the challenges given in the 2020 CSAW-ESC. We used Ghidra [1], along with the ghidra_riscv [6] plugin, as our primary static analysis tool. Whenever applicable, we relied on the Z3 theorem prover [3] to solve problems within the scope of Satisfiability Modulo Theories (SMT), such as Set A / Burst, rather than utilizing a simple exhaustive search. In situations where Z3 had trouble solving models we constructed for challenges, we either narrowed down the model to a more efficient one (e.g., fully emulating/predicting the Random Number Generator in Set B / Chase by enumerating the entire random space), implementing solution search trees optimized to efficiently search the solution space by discarding invalid branches (such as the reused-one-time-pad solver in Set C / Recycle), and when no other options worked (e.g., for Speck-like crypto in Set D / Corrupt, which has no known-plaintext attack available), we would resort to a more basic exhaustive search technique. Over the course of the competition, we developed the capability to sniff traffic over the SPI bus by soldering leads onto the relevant pins of the ESP32 chip and analyzing the traces with sigrok [9]. Additionally, we developed the novel capability of dynamically calling functions in the challenge binaries from arbitrary instrumentation code via a custom linker script and QEMU emulation, which enabled efficient offline dynamic instrumentation of original challenge code without the need to debug or instrument the real challenge hardware, or manually lift challenge functions to an emulatable form. This greatly increased the automation and efficiency of our challenge solving process, and minimized the time needed on the actual challenge hardware to solve each challenge. To further increase efficiency, we developed a set of scripts to ease deployment and interaction of the challenges on the challenge hardware, working around issues such as the unreliable Wi-Fi module, and enabling our automated challenge solving scripts to reliably function. We aimed to make all of our results as reproducible as possible with the development of automated solving scripts for challenges that required complex or precisely timed interaction, and in fact reproduced each result multiple times while creating the materials for the final submission. With this set of tools, it was possible for the NUSec CTF team to solve each challenge from the released problem sets, including the original version of Set B / Middleman.

II. Methodology

To analyze the provided firmware binaries we leveraged a combination of static analysis, on-device dynamic analysis, and emulation-based dynamic analysis. Additionally, remote work on challenges was enabled by setting up an Atomic Pi board connected to the RISC-V challenge board, and enabling the HiFive toolchains and SSH access. This allowed our team members to deploy, investigate, and attack different challenge binaries without being physically in the Northeastern University Cybersecurity and Privacy Institute lab, which helped the team solve challenges in the face of COVID-19 distancing recommendations. Additionally, we utilized pwntools [8] to automate the exploitation for certain challenges that required precise sequences of actions, as well as our own series of scripts to automate the process of uploading challenges to the board, reconnecting to Wi-Fi more reliably, and starting OpenOCD and GDB for on-board debugging. All of our utility scripts and challenge solving code is provided in the associated archive of code files alongside this report.

A. Static Analysis

In order to analyze the code contained in the provided challenge binaries, we used the open-source software reverse engineering tool Ghidra [1] and the RISC-V processor extension ghidra_riscv [6] to disassemble, decompile, and annotate the binaries. The inclusion of DWARF debug information and symbols in the binaries assisted with identifying the challenge functions of interest. Using Ghidra, we determined the architecture of the challenges, with the earlier challenges being

based on bare metal RISC-V code and the later ones being based on the open source FreeRTOS. Additionally, we installed a central Ghidra server for collaborative reverse-engineering of the challenge binaries, which was also instrumental in enabling our team to solve challenges remotely.

B. Emulation

To isolate components of challenges, such as single functions that performed verification of inputs, we created a linker script and staging code to allow loading challenge binaries in QEMU [5] user-mode emulation (`qemu-riscv32`). The script works by linking a Linux RISC-V 32-bit binary including arbitrary C code, staging code, and the full challenge binary contents extracted from the provided ELF files. The staging code then moves the challenge binary to the address `0x20010000`, and additionally maps `0x80000000` to create a memory map similar to the provided board environment. The C code can then call functions by address in the binary, which allowed for fast offline dynamic analysis without needing the board. To emulate code that invoked hardware peripherals of the challenge board, we were also able to dynamically patch the challenge binary code to skip the hardware accesses. This proved invaluable for the dynamic analysis of many of the challenges, as it was much easier to be able to run challenge code locally than debug it once it is loaded on the real challenge hardware. This script can be found in the provided source code – `esclinker.py`.

C. Hardware Probing

We were able to sniff the communication between the main RISC-V SoC and the ESP32 wifi module by probing the SPI bus connecting them. We soldered wires onto the SPI pins of the ESP32 module, and used a generic FX2-compatible USB logic analyzer and the open source Sigrok/PulseView [9] software to decode the SPI traffic and display the contents. This assisted in debugging and determining the state of some of the challenges, and facilitate the development of solutions. The setup can be seen in figure 1.

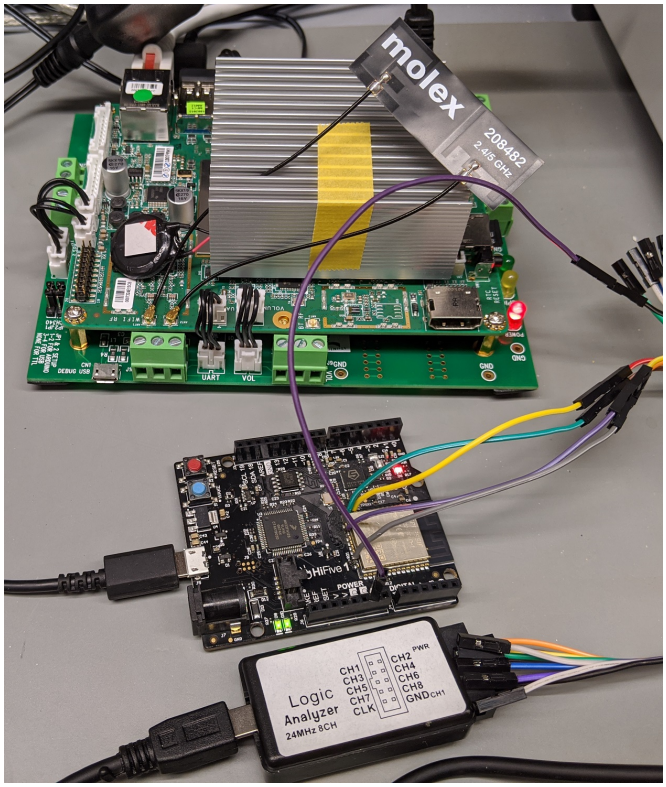

Fig. 1. Setup for remote work: Atomic Pi with attached HiFive board and logic analyzer

III. Results

A. Challenge Set A / Amnesia

In this challenge, we discovered that the `challenge()` function first compared its input to some data on the BLE partition on the Wi-Fi module via the SPI bus. If the input matches the stored data, the function returns correct, and the point score is printed. If the input does not match the stored data, the function first stores the input on the BLE partition, and then calculates the "yodel" corresponding to the input. It is worth noting that this process is time-consuming and requires a large number of operations. The function then stores the yodel on the BLE partition, overwriting the original data which it had previously written. Our solution to this challenge was to provide an arbitrary input A to the board (of at least 7 characters), and then reset the board while it was computing the "yodel" of A. By doing so, we let the `challenge()` function write A to the BLE parition, and prevent it from overwriting it with the "yodel" of A. We discovered that resetting the board does not wipe the BLE partition on the WiFi module. After the board reset, we provided A once again as input, and were presented with a message indicating we had solved the challenge. The message includes the raw AT command result containing the ESP32 module flash contents, combined with the message "Correct! Save key for report", and no other data. Therefore we have included this printout in the appendix.

B. Challenge Set A / Breakfast

By performing static analysis on the challenge binary, we discovered that the `challenge()` function accepts an input which it tokenizes using `" "` as a separator into an integer C and a string S. The function then calls encrypt on S using C as an additional argument and places the output in a buffer B. B is then compared to a static string, with the result of the comparison being used to determine if the function should return success or failure. The `encrypt()` function takes a table in the data section of the binary and calls `leftRotate()` on it parameterized over $C \bmod 0x3b$ and a constant $0x1a4$. `encrypt()` then proceeds to iterate over each byte BB of the input string, using BB to calculate an index into the rotated table for which the value is concatenated to B. The `leftRotate()` function accepts three arguments: a buffer P, C, and an additional integer Q. `leftRotate()` calls `leftRotatebyOne()` C times using P and Q as

inputs. `leftRotatebyOne()` shifts the value in its buffer Q times in 7 byte chunks in the left direction, wrapping around when needed. Thus, in order to solve this challenge, we needed to discover the correct C and S which would result in the value `challenge()` would compare the encrypted S to. In order to do this, we wrote a python script which performed the inverse of the `leftRotate()` function on the comparison string over all $C \bmod 0x3b$. By examining the resulting values, we discovered both the correct C and S to solve the challenge. The script is provided in `breakfast.py`.

C. Challenge Set A / Burst

We analyzed the `challenge()` function in this binary. We discovered that the function tokenizes its input into three integers, and then applies a set of restrictions the violation of which would cause the challenge to return failure. In order to solve this challenge, we utilized the Z3 theorem-prover via the bindings provided by claripy in angr to solve for the correct input. The code is provided in `burst.py`.

D. Challenge Set A / Flood

We analyzed the `challenge()` function and noted that it is intended to calculate the number of primes between 1 and its input, an integer, X. We discovered that the procedure converting the input string to an integer type was nonstandard, and found that while it disallowed inputting more than four digits or any input where any byte was greater than $0x39$, it did not forbid any input character below $0x30$. The implementation of this function subtracts $0x30$ from each input byte as part of the conversion process from a string of digits to an int, thus leading to an underflow for all character values below $0x30$. By leveraging this bug, it is possible to input a string which causes the function to try and calculate the number of primes below an extremely large number well outside the restrictions on user input. Additionally, we noted that the flag is only printed when the global `flood_len` is 0, and that it is set to such by the `wdog_handler()` function which is scheduled to run in `challenge()`. `wdog_handler()` sets `flood_len` to 0 if the length of data received over the SPI bus is greater than $0x400$. From the above behaviors, we deduced that by leveraging the integer underflow in the conversion procedure (and thus forcing the watchdog to run), we could send a significant amount of data over the SPI bus and cause the watchdog to set `flood_len` to 0 and cause the flag to be output. This deduction proved to be correct.

E. Challenge Set A / Parthenon

We noted that a string `ctxt` was printed during application startup. To analyze this ciphertext, we leveraged CrypTool [4], an open source tool for analyzing ciphers. From frequency analysis, we discovered that this string was likely a columnar transposition cipher due to its frequency distribution almost matching that of English. We utilized CrypTool's transposition cipher cracker set to brute force a key of maximum length 8 set to R-C-C mode and to utilize the log2 probabilities over N-grams as the cost function. This setup successfully retreived the plaintext message. This string, when given to the program, produced a response indicating that is was the correct input to solve the challenge. From a theoretical standpoint, it is possible to identify a columnar transposition cipher from the fact that its frequency distribution in no way differs from that of the language of the message (in this case, English). From this information, it is possible to automatically crack the cipher by guessing the correct number of columns and their arrangement, and ranking the resulting plaintext messages by calculating their probability from the set of bigram probabilities in the source language; whichever candidate plaintext has a higher probability (given a sufficiently large ciphertext), is the most likely. This process can be made more efficient by employing hill-climbing with simulated annealing or similar techniques.

F. Challenge Set B / Chase

Analysis of this challenge in Ghidra revealed that the challenge binary switches between different IP addresses and ports according to a random number generator seeded with the board's tick count upon initial connection, which is somewhat unpredictable due to ticks happening at a rate of 1000 per second. Initially, the IP address and port that the challenge moves to are printed to the console, but in subsequent iterations they are hidden and it has to be guessed where the challenge server can be reached next. After six iterations of this address and port hopping, the challenge is solved. Unfortunately, every time the challenge moves addresses and ports it also fully resets the ESP32 module, which we had already determined was fairly unreliable and frequently caused failed Wi-Fi connections and TCP connection refused errors when attempting to access challenges. This was very much exacerbated by the challenge requiring the Wi-Fi to work 6 times in a row, which statistically was simply not very likely. Therefore, while we did solve this challenge, due to the sheer unreliability of the ESP32 Wi-Fi it is not extremely reproducible, and you may have to run our script many times before the ESP32 decides to cooperate. The script functions by generating the entire space of random numbers that the RNG in the code is capable of generating – since it uses a multiply-add-modulo technique we can create the full repeating sequence of random numbers from the generator, and reference the sequence to guess where the generator state is currently located. The script utilizes the initial IP address and port information, as well as the different languages used for the greetings to guess candidate locations in the RNG space, and once uniquely narrowed down it is able to completely replicate the board RNG state and perfectly guess each next IP address and port (we're confident that the Wi-Fi failures we are seeing are not associated with any bugs in this script, since they have also shown up in nearly every other challenge as well). The script repeats the guessing and reconnecting process until the challenge is won, assuming of course the ESP32 doesn't randomly fail at one of the points. The script is included in `chase.py`, with necessary additional data extracted from Ghidra analysis in `chase.dat`.

G. Challenge Set B / Esrom

The `challenge()` function accepts a string input S and calculates its digest D using the `SHA2()` function (this function is not actually an implementation of SHA2). D is then compared to a digest stored in the binary, and if they match, then the function returns success. If D does not match, the function proceeds to interact with an LED on the board, turning it off and on with timings determined by a global string of `" "`, `"0"`, `"1"`, and `"2"`. We assumed that this string corresponded to the desired input encoded in Morse code. We decoded the string, treating `"0"` as dot, `"1"` as dash, `" "` as the character separator, and `"2"` as the word separator (notably, in the actual binary, `"0"` corresponded to a long LED and `"1"` corresponded to a short LED which is the inverse of the real morse code). By utilizing this methodology, we successfully recovered the flag.

H. Challenge Set B / Middleman

This challenge proved quite problematic. Upon booting, the challenge binary prints several message/signature pairs corresponding to an ECDSA scheme based off of a custom curve, the details of which are also printed, over serial. The challenge asks that we forge a signature for a given message without telling us the secret key. ECDSA signatures are composed of two components: R and S. R is the x-coordinate of a curve point $k * G$ with k being a cryptographically random number chosen *per message*. By analyzing the binary, we discovered that the `getRandomNumber()` function always returns 4. Additionally, we note that the R value in all leaked signature pairs is the same, implying the same k value was used to generate each signature. This behavior is quite dangerous under ECDSA, since an attacker having knowledge of the k value for a message/signature pair allows them to easily calculate the private key used to generate said message by the equation $\frac{S*k-Z}{R}$, with Z being the most significant n bits of the hash of the message under some hashing algorithm and n being the order of the base point G of the curve in use. Thus, the faulty random number generator present in the challenge binary would indicate that k was 4 for all of the message/signature pairs. This assumption proved to be correct. Unfortunately, we were unable to discover the correct hashing algorithm used to generate the proper signatures. Without knowing this algorithm, it was impossible for us to calculate Z for any message. Given that the integer order of G was 8, the value of Z is restricted to $[0, 255]$. Through brute forcing all possible Z values for a known message/signature pair and our knowledge of k, we discovered the private key used to generate the messages was 130. Unfortunately, even with knowledge of the private key, we were unable to generate signatures due to still lacking knowledge of the hash function. We attempted to check all common hashing algorithms, along with the custom algorithms in the `SHA2()` and `SHA1()` functions against the examples printed to the console in the challenge, but none of these produced the correct lower bits for all of the examples given. Thus, in order to solve the challenge, we developed a custom linker script capable of injecting the challenge binary into a C program run in the usermode QEMU (c.f. `esclinker.py`). By using this program, we developed a brute-forcer for the `SHA2()` function in the binary which allowed us to find the correct challenge input. At first, we assumed that the public key for the forged signature should be the same as that used when calculating the leaked signatures. This assumption was incorrect. We also assumed that the input format would be the same as the provided examples, which included parentheses and commas. This assumption also proved to be incorrect. Thus we moved on to analyzing the `SHA2()` function in the originally-released challenge binary, and we found that it broke the message up into 4 chunks of approximately equal length and concatenated the results of calls to the `SHA1()` function in its output buffer. We noted that the desired output for the first chunk of the message was quite different from the expected output, indicating that the public key it was expecting was not the one in the sample message/signature pairs. Having learned this, we modified our brute-forcer to brute force the hash one chunk at a time, thus allowing it to derive the expected challenge input. By using this method, we recovered a signature `2202 1168 125 104`, which was accepted by challenge binary. As demonstrated by the fact that the challenge was later updated, this was an unintended solution.

I. Challenge Set B / Middleman.UPDATED

The updated version of this challenge proved to be significantly different from the original. It is notable that the hash function was no longer vulnerable to the attack which allowed us to brute force its input in four discrete chunks. Additionally, the program strips all instances of parentheses and commas from the input string and replaces all sequences of such with a single space so as to normalize the input to the form of `"%d %d %d %d"`. We assumed that the values of this input correspond to the x coordinate of the public key, the y coordinate of the public key, R, and S, respectively. Given the nature of ECDSA and the faulty random number generator, x, y, and R are known values. Additionally, S is bound to $[0, 131)$. Again, we were unable to discern the correct hash algorithm to utilize to calculate Z for a message. Taking the provided hints into account, we consulted FIPS 180-4 [7] for the list of approved hash functions for cryptographic uses (this is also referenced by FIPS 186-4, the elliptic curve Digital Signature Standard), and were still unable to find any hash which produced message digests for which the provided examples of S would be valid, which left us once again unable to construct signatures despite having obtained the private key by analysis of the provided example signatures. Therefore, we tweaked our previous brute-forcer algorithm to account for this knowledge to determine the correct value for S since we still didn't know the correct hash algorithm to use when calculating Z under the attack discussed in the previous section. With this we obtained the signature `1341 1979 125 97` which was accepted by the challenge. The code for this is provided in `middlemanupd2.c`.

J. Challenge Set B / Quiz

This challenge prompted the user to answer a series of math questions presented as English text. We implemented a parser which accepted a list of tokens as input and computed the correct response for those given tokens. This challenge proved potentially more troublesome than intended due to the fact that the Wi-Fi communication from the board ended up fragmenting the challenge prompts in a highly unpredictable and inconsistent manner (randomly cutting off and/or mixing different parts of the message), which prevented them from being parsed by the script. The solver is included in `quiz.py`

K. Challenge Set B / Sequence

We analyzed the `challenge()` function for this binary. We note that it tokenizes the input by `" "` and converts all tokens to integers via calls to `atoi()`. It then compares these integers to values stored at certain indices in the "sequence" global. This variable is populated by a call to `fibbonocci()` (which interestingly, does not implement the Fibonacci sequence). In order to solve this challenge, we ported the decompilation of this function to a C program and printed the indices (24 and 28) in question, resulting in the input that the challenge binary accepted: `0 38`. Code is included in `sequence.c`

L. Challenge Set C / Game

We noted that this challenge requires the user to play a game against a dealer until the user either wins or runs out of money. The game is initiated when the user enter the `play` command over SPI. Other than this command, the user has no control over the game. The `challenge()` function calculates a seed for the random number generator upon entry based on the current time in seconds since the challenge was started, and this seed determines the output of the game. The challenge helpfully prints an initial seed value when starting a game that can be used to recover the board's tick counter used to determine the number of seconds. The game is won when the player obtains at least $\$1,000,000$, but the outcome of each game is determined by the RNG. We utilized the same linker script as detailed in the Middleman section to brute force a seed value for the RNG which would result in enough won games for victory, and from this seed value determined the correct number of seconds from board startup (around 145) to wait until starting to game to ensure victory. We developed a program which would control the board and wait the precise amount of time prior to starting the game to achieve success. We have included the code files for the solution `game.c` and `game.py`.

M. Challenge Set C / Hue

We analyzed the challenge function for this challenge in Ghidra, and noticed that it was setting the LED color based on an array of data in the binary. We extracted the array and rendered an image out of the data matching pixel colors to the way values were decoded into LED outputs by the challenge code. After guessing various possible dimensions until patterns in the image seemed to line up, the flag text was revealed as text in the image data. The decoder script `hue.py` is included.

N. Challenge Set C / Recycle

This challenge is the classical example of the repeated use of a one-time-pad. Our solution involved querying the challenge for a number of distinct ciphertexts. From these ciphertexts, we generate a tree over all possible plaintexts of the desired length padded with spaces, and utilize our knowledge of the input space (a known set of words from a wordlist extracted from the binary) over multiple instances of the repeated key and the key's length to reduce the key space down to a single value. This method works due to the fact that for any instance of a character in the ciphertext c at a specific index i, the corresponding character in the keyspace k must, when xor'ed together with c, produce a valid character in the messagespace m at i. From this methodology, it becomes possible to efficiently eliminate possible keys and converge upon the correct one as long as enough ciphertexts are known. The solver script `recycle.py` is included.

O. Challenge Set C / Virtual1

Analysis of the challenge shows that the challenge is implemented as a basic instruction emulator with instructions for adding, subtracting, and multiplying values in 4 registers, which are initialized to the values 0, 1, 2, 3. The challenge is completed when the value of register 2 matches the value of register 3, therefore we provided as input a subtraction instruction to subtract the value of register 1 from register 3: `1 3 3 1`

P. Challenge Set C / Virtual2

This challenge is an extension of the previous Virtual1 challenge, with more instructions available. New instructions included bitwise operations, and reading and writing registers to an additional memory array. The goal was to get the value of memory location 9 to be equal to memory location 8 and register 2. We implemented this with 5 instructions, first multiply register 3 by itself to produce the value 9, write register 2 to the memory location specified by register 3, subtract register 1 from register 3, and write the value of register 2 to the memory location specified by register 3 again, then exit. This was encoded as `2 3 3 3 7 3 2 0 1 3 3 1 7 3 2 0 8 0 0 0`. A simulator we used to debug this solution is provided in `virtual2.rkt`.

Q. Challenge Set D / Corrupt

There were multiple files provided for this challenge: `corrupt.elf`, `corrupt.hex`, and `corrupt.heks`. We loaded all the files into Ghidra and determined that the first 2 roughly matched, but the third one was a different program. Additionally we noticed that Ghidra produced checksum mismatch warnings on the third file. Static analysis in Ghidra revealed that `corrupt.heks` writes a string to flash onboard the ESP32 module – `Privilege Level: 0 Key: 29019203 1B3t9` . The main program in `corrupt.hex`

then reads back this value and uses the privilege level and key to attempt an encryption operation consisting of a 10-round add-rotate-xor cipher which looked similar to Speck [2]. Because of this, we didn't attempt any sort of known-plaintext attack, instead opting to run a simple brute force of the 32-bit key using the `esclinker.py` script. The key was recovered as `575703170` and additionally we needed to change the Privilege Level to `1`. After making these patches to the string sent by `corrupt.heks` with Ghidra we exported a modified version using Ghidra's export functionality, and ran it on the challenge board. Then, we ran `corrupt.hex` which read back the new values and successfully printed out the flag.

R. Challenge Set D / Impact

Analysis of this challenge revealed that the challenge function chose a random number and then required an input string with a hash with the lower 24 bits equal to the hash of that number represented as ASCII. The hash function was called `SHA1`, but despite this the algorithm looked nothing like the standard SHA1 algorithm, instead returning a 32-bit hash value and constants like `0x6e696265` (in ASCII that's "ebin", part of the "egg" series for the Pursuit challenge). We created a C file compiled with `esclinker.py` to find a matching input value by simple brute force, and a python script to automate the process of receiving the random number generated by the board and calling the emulation code to produce the hash collision. We provide the files `impact.py` and `impact.c` as part of the solution.

S. Challenge Set D / Moonlanding

Moonlanding required multiple stages of inputs to solve. First, by static analysis we determined that a string of 256 characters needed to be sent over Wi-Fi. This triggered a second stage of code that started to accept raw AT commands for the ESP32 module over UART. The second stage also resets the Wi-Fi network to a secured WPA2-PSK network and an unpredictable password. Finally we determined that the code was looking for a specific date and time via NTP – initially we tried the dates and times of the Apollo 11 launch and moon landing, as per the theme, but these didn't work. Therefore we brute forced the result it wanted using `esclinker.py` and determined it was looking for `Wed Dec 31 20:09`. We adapted an open source NTP server implementation in python from GitHub to always return this date and time, and used the following AT commands to change the wifi back to unsecured and enable the NTP client to connect to the server: `at+cwsap="ESC-2020","",5,0`, `at+cipsntpcfg=1,0,"192.168.4.2"`. (We set a static IP `192.168.4.2` locally to avoid the error-prone and unreliable ESP32 DHCP process during every Wi-Fi connection.) However, during this stage we struggled with the reliability of the challenge code, each command needed multiple tries until it was accepted (therefore we were not able to produce a script that can automatically solve the challenge, it needs humans present). Next, entering `DEBUG MODE OFF` exits the debug mode and triggers the NTP date and time check, which will print the flag to the next TCP connection to the challenge server. The code files `ntpserver.tar` and `moonlanding.c` are provided.

T. Challenge Set D / Pursuit

When connecting to this challenge over the Wi-Fi, the following text is printed

```
Greetings! You are connected...

Y2FuIHlvdSBmaW5kIG1lPw==
Enter String:
```

The base64 string in this output decodes as `can you find me?`. Previously we had identified some debug symbols in the other setD binaries called `egg_1` - `egg_7`, which corresponded to character arrays containing fragements of text – when put in order, these formed the URL `https://pastebin.com/wCdPGYah`, which when accessed contained the text `How did you find me?! The answer you are looking for is "345732_399_hun7"`. We attempted to input the provided flag in this text, and it was accepted by the challenge.

IV. Conclusion

In conclusion, the NUSec CTF team was able to utilize a number of open-source and in-house developed tools in order to solve all released challenges in the 2020 CSAW-ESC. Over the course of the challenge, we developed tools and methodologies to perform both static and dynamic analysis along with dynamic instrumentation. Furthermore, even though we did not need to utilize it to solve a challenge, we developed the ability to inspect traffic over the SPI bus. Across all challenges, the primary difficulty we experienced was not inherent to the challenges themselves, but rather it was related to the viability of the challenge platform. We would often find that challenges would fail to boot after flashing them to the board (e.g., amnesia) or that the board would randomly refuse connections to the specified port for unknown reasons. Because of these issues, it was often the case that our solver scripts would run in a non-deterministic fashion and would only sometimes achieve success due to difficulties interacting with the platform. In order to somewhat mitigate these issues, we developed and integrated a high level of fault tolerance into our tooling for interacting with the platform. In addition to this hurdle, it was somewhat difficult to determine what output of certain challenges constituted a flag due to a lack of a standardized flag format. It is apparent by looking at our appendix that the format of flags and solutions that they wildly vary in format. Despite these issues however, we were able to solve every provided challenge as well as the original version of the Middleman challenge.

REFERENCES

[1] N. S. Agency. Ghidra. https://ghidra-sre.org/, 2019. [Online; accessed 28-October-2020].

[2] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. https://eprint.iacr.org/2013/404.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] C. developers. Cryptool. https://www.cryptool.org/en/, 2020. [Online; accessed 29-October-2020].

[5] Q. developers. Qemu. https://www.qemu.org/, 2020. [Online; accessed 28-October-2020].

[6] mumbel. ghidra_riscv. https://github.com/mumbel/ghidra_riscv, 2020. [Online; accessed 29-October-2020].

[7] NIST. Fips 180-4. https://doi.org/10.6028/NIST.FIPS.180-4, 2014. [Online; accessed 29-October-2020].

[8] pwntools developers. pwntools. https://github.com/Gallopsled/pwntools/, 2020. [Online; accessed 28-October-2020].

[9] sigrok developers. sigrok. https://sigrok.org/wiki/FAQ, 2020. [Online; accessed 29-October-2020].

V. Appendix: Solutions

Set	Challenge Name	Input/Flag	Points
A	Amnesia	+SYSFLASH:64,⟨asdfasBBBBBB*BBBBBBBBBBB0200AAAA AAAAAAAAAAAAAAAAAAAAAAAAAg]Z	130
A	Breakfast	4 ItsNotBaconian	100
A	Burst	3880960 2209 6506496	50
A	Flood	*SaveforReport?TarbelaItaipu?Hoover?AtatuRK	100
A	Parthenon	youmeantotellmethatmyclassicalcipherisntsecure	70
B	Chase	1*4mH3r3	100
B	Esrom	remorseful	70
B	Middleman	2202 1168 125 104	(150)
B	Middleman.UPDATED	1341 1979 125 97	150
B	Quiz	125220C-61595A+	130
B	Sequence	0 38	50
C	Game	D0u8l3d0x4+Wice;e2 1191 271567	150
C	Hue	THINKCOLORFULLY	150
C	Recycle	Flag:Uoˆ/’B⟨ 4502d372142004e4c48106358594821334e0f5a444b0b6c72	100
C	Virtual1	1 3 3 1	50
C	Virtual2	2 3 3 3 7 3 2 0 1 3 3 1 7 3 2 0 8 0 0 0	100
D	Corrupt	f149=1 h4v3 bin REst0red!!	200
D	Impact	.Flag:Yˆ-Wf’⟩H-KW2x⟨N	150
D	Moonlanding	AT+Fl4g=oThisDaFl4gGYFl4gs	250
D	Pursuit	345732_399_hun7	100
	Total		2200