IOActive®
Research-fueled Security Services

\ WHITEPAPER \

Shuffle Up and Deal: Analyzing the Security of Automated Card Shufflers

Joseph Tartaro
Principal Security Consultant
Enrique Nissim
Principal Security Consultant
Ethan Shackelford
Associate Principal Security Consultant

version 1.0
August 2023

Table of Contents

Introduction

IOActive, Inc. (IOActive) conducted a comprehensive analysis of the security aspects of ShuffleMaster's Deck Mate 1 (DM1) and Deck Mate 2 (DM2) automated shuffler machines. Primarily used at poker tables, these machines are widely adopted by casinos and cardrooms and are commonly used in private games. While the primary objective of these devices is to enhance game speed by assisting dealers in shuffling, they also ensure security through various deck checks, and their control over the deck renders them highly desirable targets for attackers.

IOActive's research answered the following questions:

1. Is cheating possible if one of these hardware devices is compromised?
2. How feasible is it to perform such an attack?
3. What can be done to prevent and/or mitigate the risk of cheating?
4. How can players and gaming operators protect themselves from this kind of cheating?

It is worth noting that no signs of code from the manufacturer performing any malicious or hidden functions were found in either of the audited shufflers. Different groups across the internet have speculated that shufflers contain secret logic that Casinos and/or card rooms could leverage to cheat players or increase house edge. Having thoroughly reverse engineered the entire state machine of the original firmware for both shuffler models, we found no evidence whatsoever that this was the case.

Two factors that play a crucial role in determining the risk of cheating are: (i) the machine's capabilities and (ii) the attacker's level of access. This document describes the hardware architecture and software components of the DM1 and DM2 and presents five plausible attack scenarios. Ultimately, IOActive demonstrates how a compromised card shuffler could be used to reveal the full order of the deck at all times and how this information could be leveraged to gain an advantage. Additionally, we show how a compromised shuffler could be manipulated to put cards in a specific order, which could then be exploited to force wins or losses, and even trigger jackpots and large cash payouts to the whole table.

IOActive describes the feasibility of the proposed attacks in various arenas where automated card shufflers are used, including a number of attacks that can be performed remotely (depending on device's configuration) or with the cooperation of an insider: two attack vectors commonly considered to be high likelihood. In addition, we will explain how a player at the table (i.e., an actor with external physical access to the device) could trivially perform attacks without detection.

The environment where automated shufflers are used is of extreme importance. Casinos have established a comprehensive framework consisting of communication protocols and standards that govern the behavior of electronic gaming machines (EGMs). For this reason, IOActive thoroughly examined and reviewed the security aspects of the following specifications:

1. Gaming Authentication Protocol (GAT) v4.2
2. Network GAT Interface Specification v1.1
3. Gaming to System (G2S) Message Protocol v.1.0.3

Note that all of the vulnerabilities that IOActive found in the Deck Mate shufflers and the weaknesses identified in the specifications were communicated to the manufacturer. IOActive also informed some large gaming operators of physical mitigations that they can implement to help reduce the likelihood of an attacker exploiting these issues.

Motivation

This work started after the controversial hand played during the "Hustler Casino Live" stream on September 29, 2022. An official report from the Hustler Casino was published[1] with a section named "Cybersecurity and Technological Audit" that presented the following table.

Potential Attack Vectors	Estimated Priority	Estimated Complexity
Table	Low	Complex
RFID	None	Highly Complex
Card Shuffler	Low	Highly Complex
Production Booth and Operations	High	Easy
Network, PC Workstations, And Systems	High	Easy
Communications	Medium	Complex

Figure 1. Areas examined after controversial Huster Casino live stream

The potential attack vector that caught IOActive's attention was the card shuffler, as it was marked as highly complex and thus received a low priority for the investigation. However, the details released in that same report describing the investigation of the shuffler, while likely adequate in the context of the cheating scandal in question, did not appear exhaustive when considering the overall security of the device.

CARD SHUFFLING MACHINE

Bulletproof inspected the card shuffler for tampering and the presence of a remote eavesdropping device. The shuffler is a DECKMATE 1® manufactured by Shuffle Master. The general rule in cybersecurity is if a device knows the value of an object, then an attacker can know that object's value. The DECKMATE 1® cannot read any information about the cards other than the number of the cards in the deck. It is therefore concluded the DECKMATE 1® is safe for HCL poker play. Bulletproof recommends tamper-evident tape also be used to seal the case and inspected before each game by the dealer or any HCL staff as an additional security measure.

Figure 2. Entirety of Huster J4 Report Shuffler Investigation Notes

In addition, one of the statements from the conclusions of the audit was that the shuffling machines were secure and cannot be compromised:

BULLETPROOF'S CONCLUSIONS

Key findings include: 1) The Deckmate shuffling machine is secure and cannot be compromised; 2) It's extremely unlikely that any card-reading device could have been stored in a water bottle or other object on the table; 3) RFID technology used by "Hustler Casino Live" is safe. Any device that intercepted a signal would receive a serial number, not the actual card; 4) Radio communication to the on-floor camera operator is not an issue; 5) The PokerGFX system was free and clear of malware, installed programs or systems that could intercept hands.

Figure 3. Hustler Report Conclusions on Deck Mate Shuffler

These details led IOActive to initiate a more thorough investigation of the shufflers, and the various standards surrounding their use.

[1] https://hustlercasinolive.com/j4report/

Scope of Research

IOActive acquired both DM1 and DM2 shuffler devices from the secondhand market. The DM1 shuffler came with two separate EEPROM chips: a TMS27C512 with software version 1.12.002 and a M27C512 with software version 1.11. The DM2 shuffler has two main components, a Linux runtime environment running on an NXP iMX.28 which loads firmware from external NAND memory and an NXP LPC1769 ARM chip with firmware stored on internal flash memory.

The DM2 included a software package bundle with the following encrypted installation files:

- CardRec v5.0.23
- NXP v1.0.118
- Support v1.0.006
- UI v2.0.171

The DM2 itself had the following versions installed and running:

- Camera_FPGA_Primary v0.1.028
- CardRec v5.0.037
- DeckLib v1.3.128
- DeckMate2_FPGA_Primary v0.0.011
- Games v1.0.095
- NXP v1.0.172
- Production v2.0.012
- Support 1.0.018
- UI v2.0.254
- DisplayModule_SysPrep v1.0.021

The research primarily focused on the DM2 shuffler, as it is the latest version used in card rooms and contains an internal camera for security. IOActive also investigated the DM1 device and explored potential cheating avenues, as well as analyzed the international gaming standards used by the gaming industry.

Impact

The vulnerabilities identified as part of this research allow attackers to compromise electronic shuffler devices and use them to aid in cheating. IOActive primarily focused on the shufflers used in poker, although variations of these shufflers are used in casino table games, such as blackjack and baccarat, which can put gaming operators at risk. The increased ability to cheat at poker allows attackers to take money from other players via standard poker stakes as well as target casino jackpots, such as “Bad Beat” jackpots.

Cheating can be performed by a rogue insider, such as a corrupt employee, players physically present at poker tables, or remote players via network capabilities. Full compromise of the DM2 shuffler gives an attacker the ability to not only sort the deck, but to always know the state of the deck, meaning they know what each player holds in their hand.

Due to the standard setup of poker rooms, this attack can occur even in heavily surveilled rooms, as the cameras typically cannot see under the poker table. The issues IOActive found allow for full device

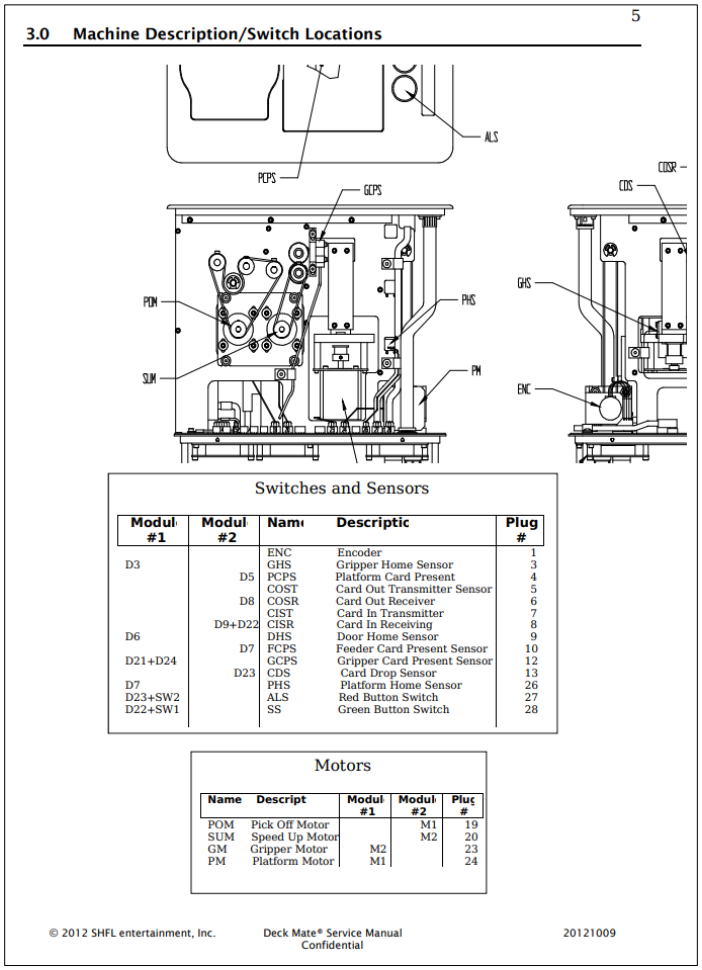
compromise and affect the inherent trust that poker players must put in gaming operators to protect the integrity of the game.

Attack Scenarios

This section sets forth five realistic attack scenarios that are possible because of IOActive's findings. They are based on real-world observations of environments where ShuffleMaster Deck Mate devices are in use or official ShuffleMaster device documentation.

Maintenance Employees

Rogue maintenance employees working on shufflers have access to the devices' internals. Referencing the ShuffleMaster Deck Mate Service Manual and the Scientific Games DM2 Participant Edition document available on the Internet, you can see the internal complexity of the shufflers.



5

3.0 Machine Description/Switch Locations

Switches and Sensors

Modul #1	Modul #2	Nam	Descriptic	Plug #
		ENC	Encoder	1
D3		GHS	Gripper Home Sensor	3
	D5	PCPS	Platform Card Present	4
		COST	Card Out Transmitter Sensor	5
	D8	COSR	Card Out Receiver	6
		CIST	Card In Transmitter	7
	D9+D22	CISR	Card In Receiving	8
D6		DHS	Door Home Sensor	9
	D7	FCPS	Feeder Card Present Sensor	10
D21+D24		GCPS	Gripper Card Present Sensor	12
	D23	CDS	Card Drop Sensor	13
D7		PHS	Platform Home Sensor	26
D23+SW2		ALS	Red Button Switch	27
D22+SW1		SS	Green Button Switch	28

Motors

Name	Descript	Modul #1	Modul #2	Plug #
POM	Pick Off Motor		M1	19
SUM	Speed Up Motor		M2	20
GM	Gripper Motor	M2		23
PM	Platform Motor	M1		24

Figure 4. ShuffleMaster Deck Mate Service Manual

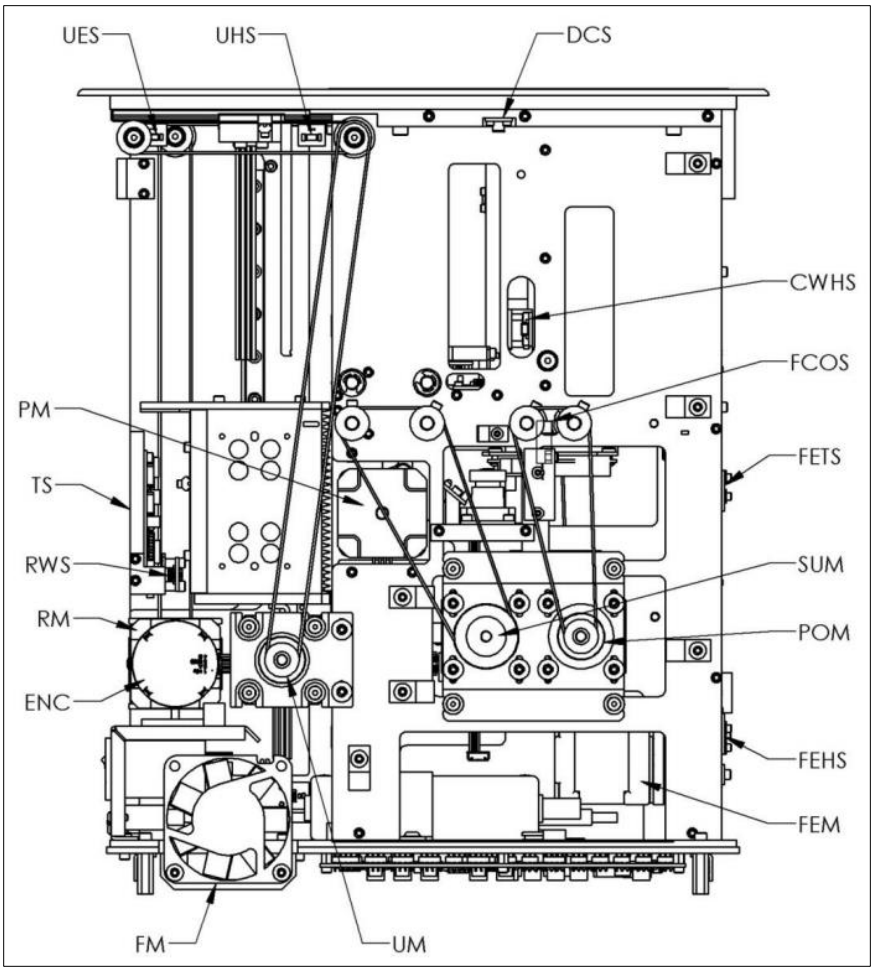


Figure 5. Scientific Games DM2 Participant Edition

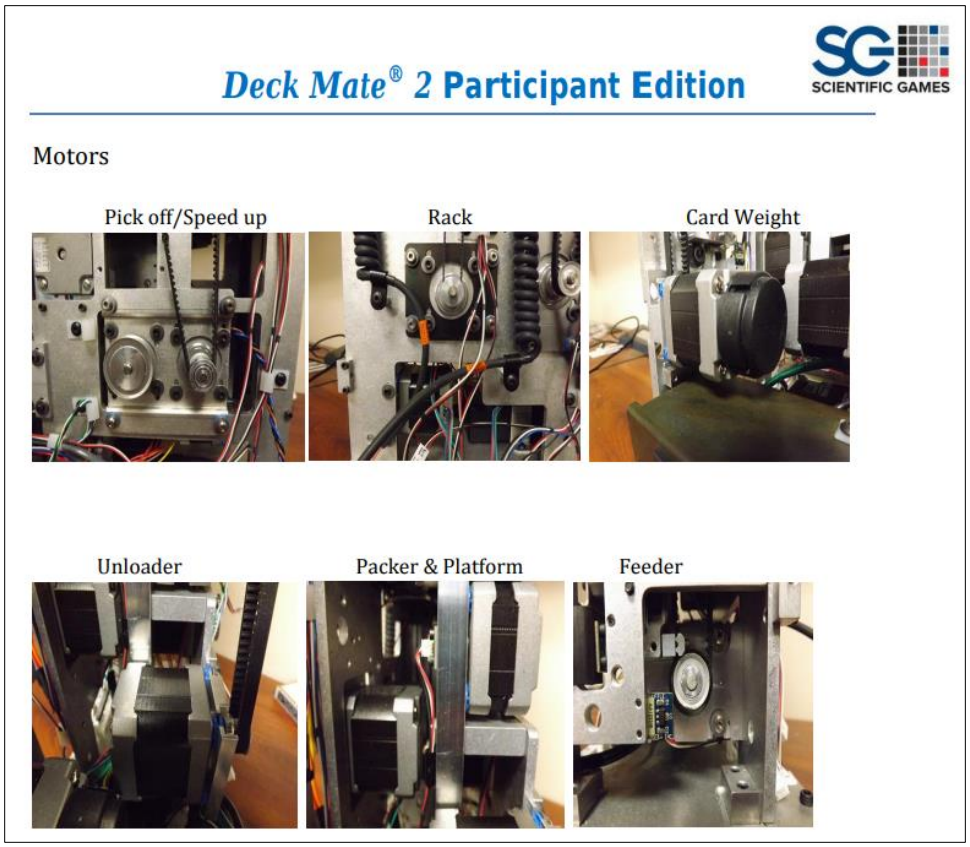


Figure 6. Scientific Games DM2 Participant Edition

The devices contain many rubber belts, sensors, and motors which require maintenance on a regular basis. IOActive's understanding is that major gaming operators who purchase these shufflers directly from the manufacturer enter into a maintenance agreement which specifies that only manufacturer employees or approved vendors can work on the devices; however, secondhand market device operators must use third-party maintenance employees as they do not have a contractual agreement with the manufacturer. In the event that one of these maintenance employees has gone rogue or is compromised, they can use their internal access to compromise the software and ultimately backdoor the devices to aid in cheating.

Gaming Operator Employees

Casino employees/device operators have unrestricted access to shuffler devices, but unlike maintenance workers, they might not have trivial access to the internals of the device; however, the DM2 has exposed Ethernet and USB ports on the rear of the device.

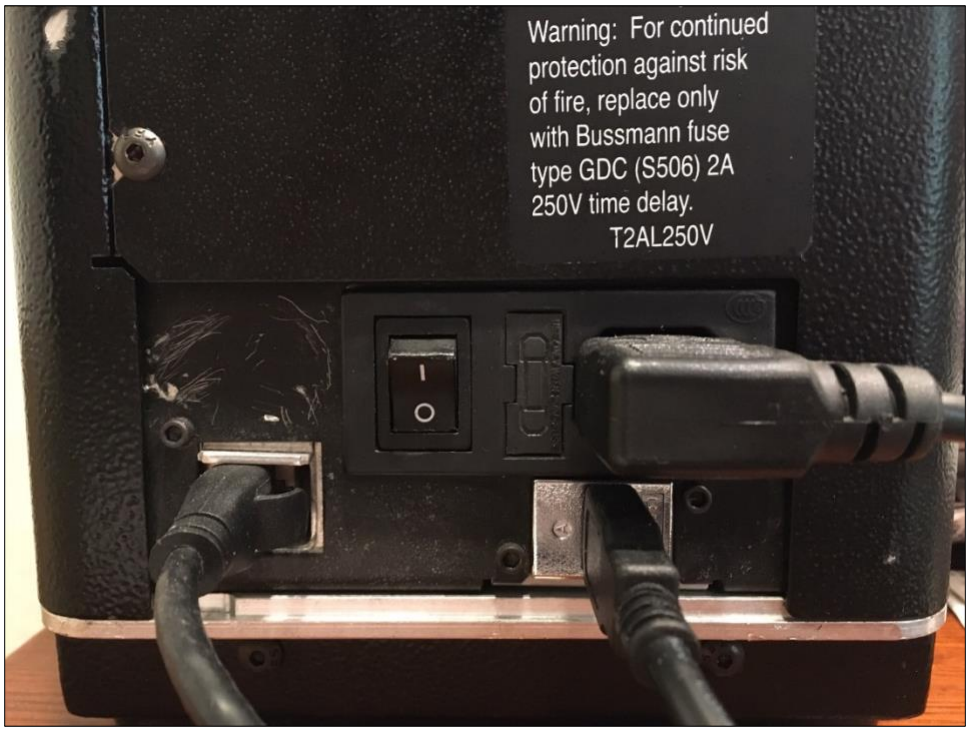


Figure 7. DM2 Ethernet and USB Ports and Power Switch

By exploiting the issues IOActive discovered on the DM2, a rogue employee or operator could gain full code execution via either of these ports. Once full code execution has been obtained, the attacker can persistently backdoor the device's firmware to aid in cheating.

Attacker at Poker Table

Unlike a standard casino table game, such as blackjack or baccarat, which may be using one of these devices, the poker environment presents a unique attack surface with external attackers that have zero ties to the gaming operators.

The standard implementation of a shuffler at a poker table, as shown in Figure 8 and Figure 9, is to physically cut a hole in the top of the table and suspend the shuffler by its metal frame.

Figure 8. Poker Table with Shuffler and Dealer Tray Cutouts[2]

Figure 9. Poker Table with DM2 Shuffler Installed[3]

The device's power switch and Ethernet and USB ports are all exposed to the players sitting at the table, out of sight of the surveillance cameras. It is very common for players to reach under the table, dropping chips,

[2] https://gorillagaming.net/gorilla-gaming-becomes-the-official-poker-table-of-the-world-series-of-poker/
[3] https://twitter.com/Kevmath/status/1132724641074515968/photo/3

storing chip trays under them, etc. A nefarious player could select a specific seat for easier access to the ports under the table and compromise the device and ultimately cheat during a live poker game.

Furthermore, it was pointed out after the 2023 World Series of Poker by a dealer at that event that not only was it possible for players to reach the ports on the shuffler, but that players were regularly and openly doing so – the USB port was being used as a convenient place to charge their phones.

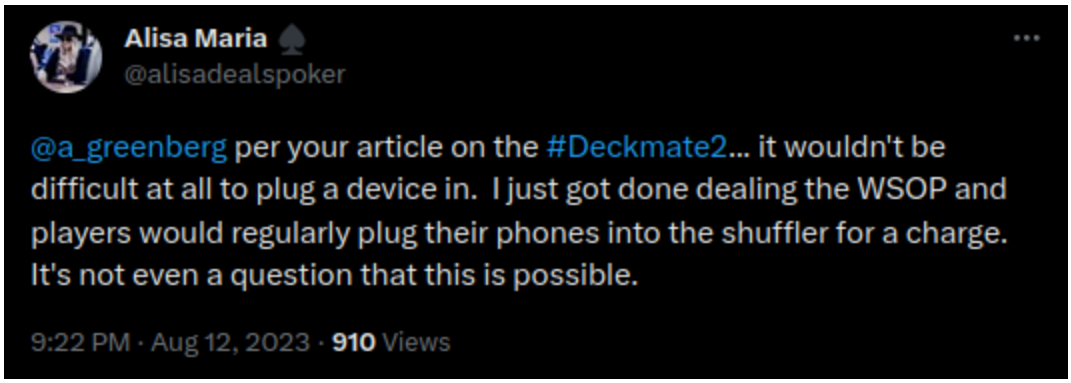


Figure 10. WSOP Dealer Describing Easy Access to DM2 USB Port[4]

Attacker with Network Access

The DM2 shuffler offers network capabilities via its Ethernet port. IOActive discovered a large remote attack surface that could allow an attacker with network access to remotely compromise the device during a poker game.

Attacker with Cellular Network Access

After analyzing the configuration files on a DM2 device and referencing official documentation and gambling commission documents available on the Internet IOActive discovered a version of the DM2 that contains a cellular modem. This appears to be used for device rental, where the device reports usage to the manufacturer via the cellular network and the operator is charged based on the number of shuffles. Only limited information is publicly available regarding these modems, and IOActive was not able to acquire one for testing.

Depending on the configuration and technology in use by the cellular modem, an attacker may be able to deploy a rogue cellular base station and force a connection with the DM2 by masquerading as the legitimate base station. Once the DM2 is connected to the attacker-controlled base station, the attacker will have direct access to the exposed network services and can exploit the issues described in this paper to remotely compromise the device over the air (OTA), potentially completely outside of the building where the shuffler is located.

[4] @alisadealspoker https://twitter.com/alisadealspoker/status/1690579436201029632

Casino Architecture and Standards

When casinos started introducing Electronic Gaming Machines (EGM), operators needed a networked environment to easily monitor and communicate with all of the devices. The devices communicated with a floor controller, which relayed information to a floor system or other backend system, and they all used their own proprietary protocols. In addition, they generally used low-bandwidth serial communication with limitations on features and capabilities. This architecture led to many issues when attempting to incorporate new devices or new business functionality, and quickly created demand for a higher bandwidth network-based communication standard.

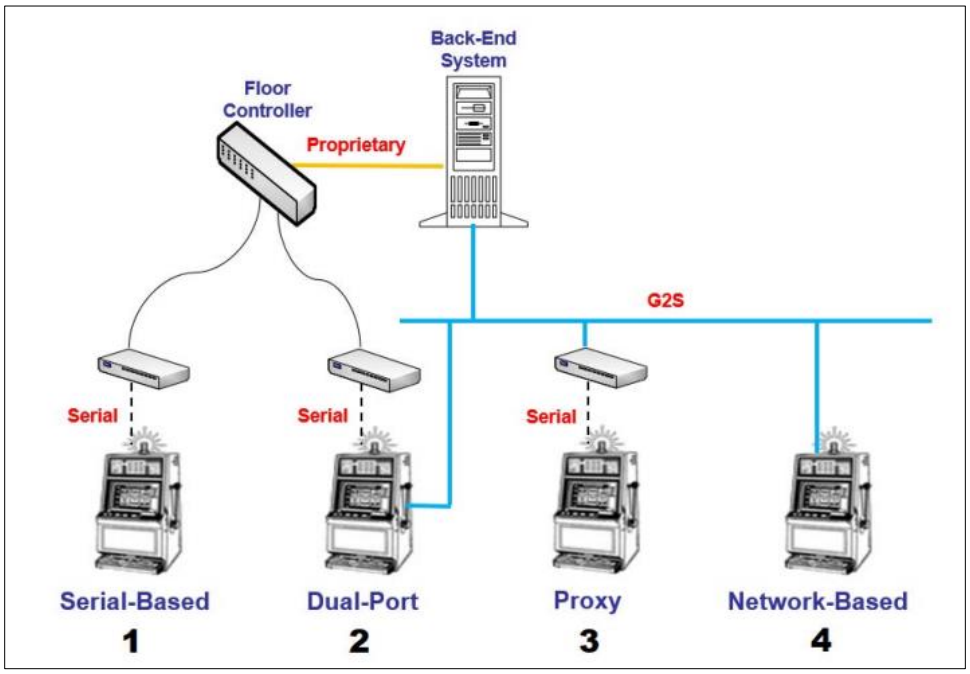


Figure 11. Traditional Casino Floor (IGSA Unleash the Power of Your Floor, 3rd edition)

As can be seen in Figure 11, devices use a variety of open and proprietary protocols.

Gaming Standards

The International Gaming Standards Association (iGSA) is responsible for the standards implemented across the gaming industry. The iGSA was founded to fix communication standards issues across EGMs and peripherals and has created an entire suite of communications protocols for gaming operations.

There are many gaming standards with specific purposes, including land-based, online, and regulatory types. Land-based standards include communication and support between the EGMs, peripherals, and backend systems on a casino network. Online standards refer to the Third-Party Game Interface (TPI), which is a standardized interface between iGaming platforms, remote game servers, and progressive jackpot controllers. Regulatory standards exist to help regulators successfully perform their duties to ensure the security and compliance of EGMs and peripherals. Their intent is to allow regulators to ensure that the software running on the devices has not been modified in any way since the production release by the manufacturer.

Game-to-System (G2S)

G2S was designed to be the primary protocol standard for network communication between EGMs and host systems, such as progressives, player tracking systems (loyalty programs), and ticket-in/ticket-out systems. A wide range of business functionalities are supported by G2S, including money handling, configuration, and software downloads, as well as other advanced functionalities.

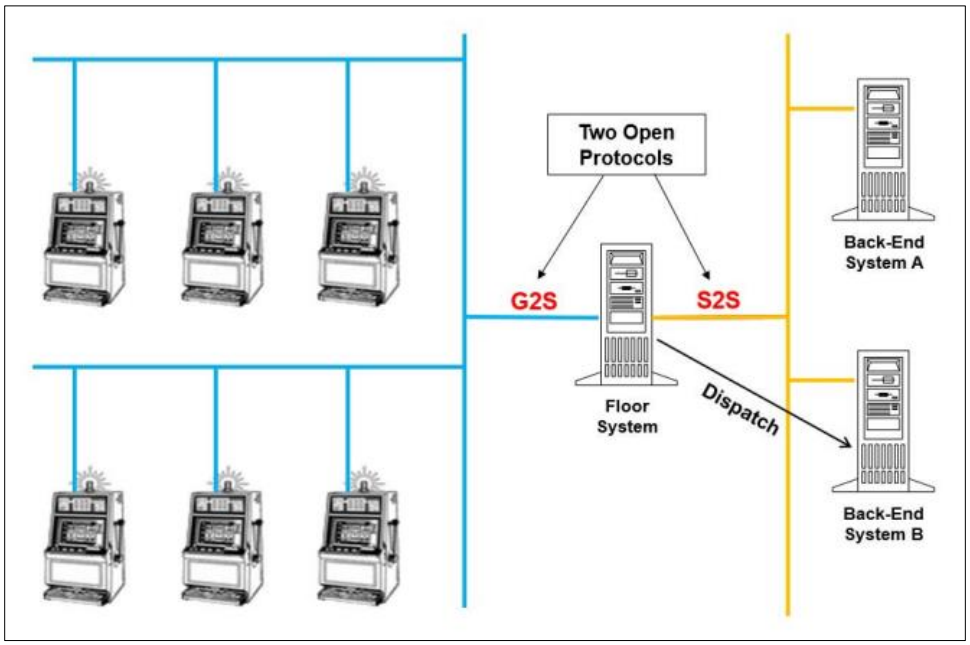


Figure 12. Modern Casino Floor (IGSA Unleash the Power of Your Floor, 3rd edition)

G2S is a high-speed networked protocol supporting TCP/IP communication channels leveraging industry-standard technologies, such as SSL, XML, and other IP protocols. To be G2S compliant, implementations must conform to XML 1.0 and XML Schema (XSD) 1.0, which define a set of required classes/functionality. Support for additional classes/functionality are considered optional and are categorized as administrative classes and application classes. Administrative classes perform functions such as automated software configuration. Application classes contain operation functionality, such as progressive bonus support and vouchers.

An example of a required class would be the regulatory class Game Authentication Terminal (GAT) which provides a set of commands to perform the software/device authentication required by casino regulators. The GAT class does not specify the authentication method to be used, although it does assume that it will be returning a value that contains authentication information (program/module identifier and hash/checksum/signature). It is worth noting that the standard considers the terms checksum and signature to be interchangeable and should not be confused with a cryptographic digital signature. GAT is described in more detail in Game Authentication Terminal (GAT).

Game Authentication Terminal (GAT)

Historically, GAT was primarily used for EGMs. In today's casino architectures, GAT is used to authenticate running software across many devices and systems, including gaming peripherals. There are multiple versions of GAT communication, the original was over a serial interface, more modern implementations over G2S, S2S XML protocols, and the most recent being the Network GAT Interface (NGI), via HTTP, REST, and JSON.

The overall concept of GAT is to verify that the software running on the device is original and unmodified. This process is done by requesting that the device calculates a hash value for the software, which can then be compared with a known value to ensure that it is correct.

GAT Process

The standard GAT process includes the following steps

1. Establish communications with the device.
2. Request the list of components for authentication, such as:
    - Operating System
    - Individual Games
    - Peripherals
3. Request authentication of components:
    - SHA1-HMAC for software
    - CRC-32 for peripheral
    - User-provided offset and seed (optional)

Serial GAT

The devices covered in IOActive's research supported serial GAT. The following table provides an overview of the standard message structure.

Table 1. Serial GAT message structure

Command	Length	Message Data	CRC
1 Byte	1 Byte	0 - 251 Bytes	2 Bytes
Identifies the command being sent	Total number of bytes in message	Contents of the message	CRC-16 of command, length and message

Four commands are supported.

Table 2. GAT commands

Request	Description	Response	Description
0x01 SQ	Status Query	0x81 SR	Status Response
0x02 LASQ	Last Authentication Status Query	0x82 LASR	Last Authentication Status Response
0x03 LARQ	Last Authentication Results Query	0x83 LARR	Last Authentication Results Response
0x04 IACQ	Initiate Authentication Calculation Query	0x84 IACR	Initiate Authentication Calculation Response

You can query the status of the device and it will return its supported GAT version, status on current calculation, last authentication results, and supported data formats. When issuing a new authentication, you are required to provide an Authentication Level, the value 0xBA indicates "Special Function" and no other authentication values are defined by the GAT specification. There are four special functions defined within GAT.

Table 3. GAT special functions

Special Function	Description
Get Special Functions	Returns a list of special functions supported by the EGM
Get File *filename*	Returns the contents of the file specified in *filename*
Component *name salt*	Returns the hash of the component specified in *name* calculated using the SHA1-HMAC algorithm and the salt value specified in *salt*
doVerification *name algorithm parameters*	Returns the hash of the component specified in *name* calculated using the algorithm specified in *algorithm* and the parameters specified in *parameters* The parameters are dependent on the algorithm: CRC16, CRC32, or SHA1-HMAC

As you can see, there is no actual cryptographic authentication of the software components, and the weak hashing mechanisms allow attackers to generate collisions or simply fake the correct hash values.

The use of an HMAC and user-supplied offset and seed values was an attempt to provide extra safeguards; however, this can trivially be bypassed by an attacker who has completely compromised the software. For example, an attacker can patch the running software and keep a dictionary of modified and original bytes in a code cave. Then, during the authentication mechanism, the attacker's modified code can replace the modified bytes for the original bytes during the hash digest and always return the correct value.

The goal of GAT is to perform what's known as 'attestation', Attestation allows a program to authenticate itself and remote attestation is a means for one system to make reliable statements about the software it is running to another system. This process requires the usage of a root of trust, a TPM, and concepts from the domain of public key cryptography. None of these are referenced in the G2S or GAT specifications.

One interesting theoretical technique leverages two of the *parameters* of the doVerification special function. The *startOffset* and *endOffset* parameters allow for an auditor to specify a particular portion of the component to perform verification of, and the hash returned by the device (whether CRC or SHA1-HMAC) will be returned for just that section of component data.

First, understand that for an 8-bit value (a byte), it is possible to build a table from 0 to 255, where each byte's corresponding hash value is recorded, a version of something known as a "rainbow table". An attacker could select a hash type, for example CRC32, then generate this rainbow table for all possible byte values. This would allow for correlation between a CRC32 returned by doVerification, and the corresponding byte.

Consider a scenario where an attacker requests the CRC32 via doVerification with *startOffset* and *endOffset* set to 0 and 1 respectively. The GAT service running on the device will calculate the CRC32 of the first byte of the component being verified, and send that back to the attacker.

The attacker now possesses a CRC32 of the first byte of the component. The attacker can then look this CRC32 value up in their pre-generated rainbow table, and learn the corresponding byte value – that is, the byte at offset 0 in the component data. The attacker may then repeat this attack with *startOffset* and *endOffset* set to 1 and 2 respectively, and thereby learn the second byte. Repeating this for the entire component allows for full exfiltration of the entire component code and data, which may allow for the leaking of secrets, or exposure of software binaries for reverse engineering.

Automated Card Shufflers

Casinos and card rooms around the world use automated card shufflers for table games, such as blackjack, baccarat, and poker. These shufflers are offered in various shapes and sizes, with some being able to handle up to eight deck shoes. There are a few major brands and models, the most popular being the ShuffleMaster Deck Mate models.

Deck Mate 1

The DM1 is the first version of the Deck Mate shuffler released by ShuffleMaster

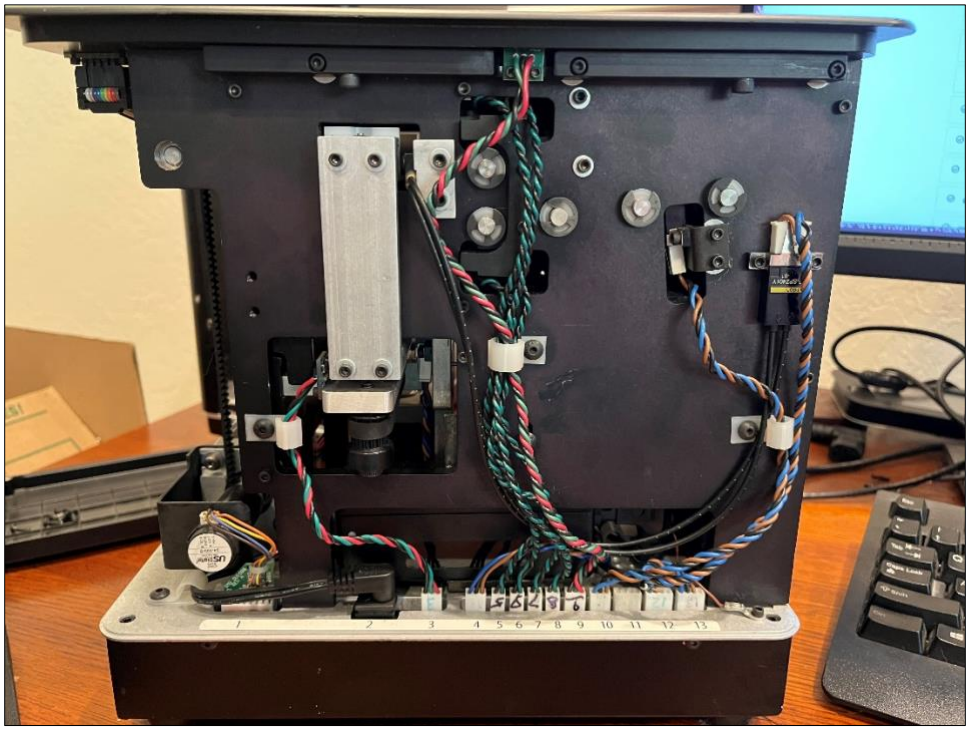


Figure 13. DM1 Right-side Internals

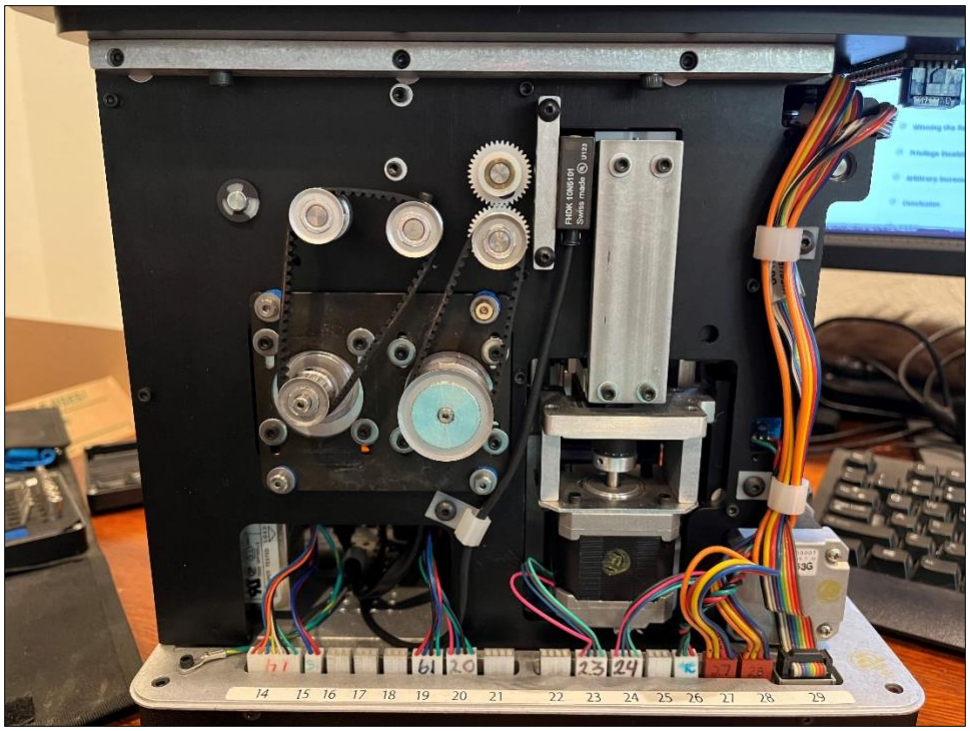


Figure 14. DM1 Left-side Internals

Serial Interface

There is an Ethernet port on the back of the DM1 which is a serial port, running at 19200 8N1.

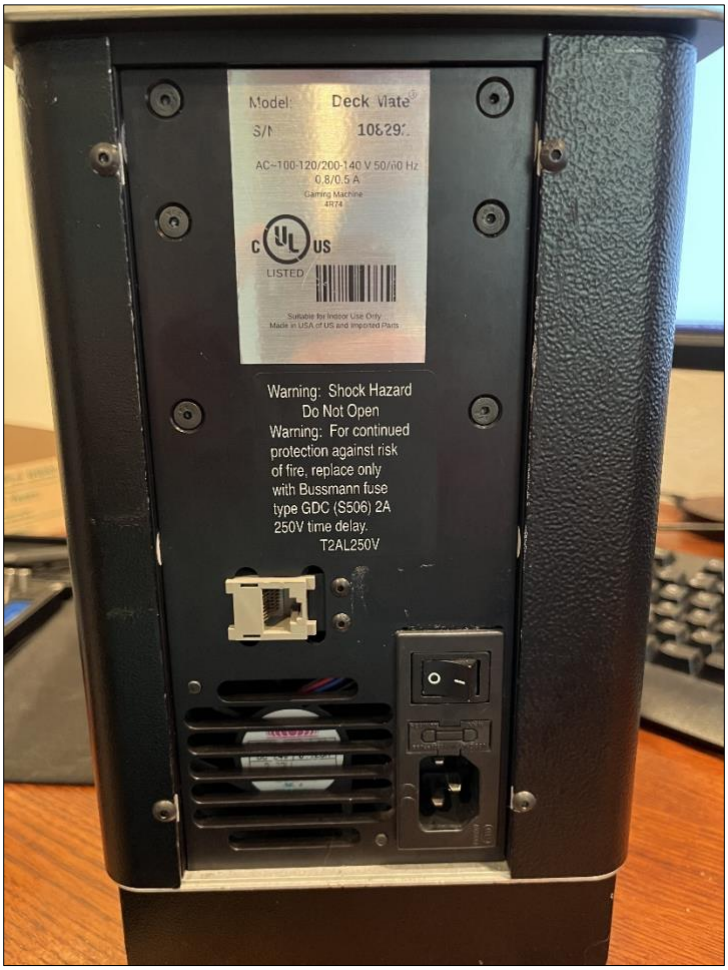


Figure 15. DM1 Rear Serial Ethernet Port

Starting the device up and monitoring the serial interface you will see the following banner:

```
\0
    SHUFFLE MASTER DECK MATE SHUFFLER
```

IOActive enumerated every possible byte with a carriage return and newline, as well as manually pressing various keys and sequences while connected to the console, and never received additional output. The only data received over this interface was via the 'Download to PC' feature described in Service Menu.

Service Menu

The DM1 contains a service menu that can be entered by holding down the green button while powering on the device. This service menu gives access to several features, such as:

- A cleaning mode which activates pick-off and speed-up motors at low speed for hand cleaning rollers
- The state of each sensor and switch
- Download to PC, which retrieves history logs by connecting to the serial interface

- Various other platform position and motor tests

Here is a truncated example of a history log retrieved using Download to PC:

```
Deck Mate Shuffler History Log.
Serial Number: 108292
Report T:10:02:39
Report D:04-29-00
Current Gripper Offset:203
Current Platform Offset:189

Powered Up   :  Time:06:58:23,Date:04-24-18,P:243,G:224,Cyc:2982
Auto Setups. :  Time:23:18:03,Date:01-11-00,P:96,G:245,Cyc:0
Powered Up   :  Time:23:19:45,Date:01-11-00,P:250,G:100,Cyc:0
Extra Card.  :  Time:23:20:26,Date:01-11-00,P:270,G:100,Cyc:0
Jam Recovery:  Time:17:12:45,Date:08-14-15,P:0,G:246,Cyc:0
Powered Up   :  Time:23:40:52,Date:01-30-00,P:82,G:246,Cyc:24
Auto Setups. :  Time:00:04:46,Date:01-31-00,P:189,G:203,Cyc:24
Powered Up   :  Time:10:07:13,Date:02-11-00,P:189,G:203,Cyc:36
Powered Up   :  Time:02:03:56,Date:02-13-00,P:189,G:203,Cyc:37
Door Opened. :  Time:07:49:54,Date:02-13-00,P:209,G:203,Cyc:190
Missing Card.:  Time:07:54:27,Date:02-27-00,P:199,G:203,Cyc:1052
Power Ups = 89
CDS Failures:0
Missing Card  = 201
Extra Card  = 4
Door Open Jams = 231
Jam Recoveries = 2
Auto Setups = 11
Platform OOP = 631
Total Cycles:1056
```

Physical Shuffling Mechanism

The DM1 contains two compartments: one for holding the initial deck and another for delivering the shuffled deck. The primary components of the first compartment, where the unshuffled deck is stored, include a card sensor and a feeder motor belt. Cards will be fed one at a time (from the bottom of the deck) into the shuffling compartment if the card sensor detects them. The shuffling compartment consists of a vertical rack with a movable platform and a set of grippers. The movable platform positions the currently processed cards to the correct location. At this point, the grippers will grip the cards at that location and the platform will move down creating a gap for the next card to be fed in. The locations where the platform needs to move and how many cards the grippers will hold are configured beforehand based on a randomized array of positions.

The DM1 supports different game configurations: blackjack single-deck, double-deck, and poker. One of its vital security features is the ability to count the cards as they are shuffled and automatically abort the cycle if the number of cards differs from the current game configuration.

When all the cards are successfully processed, the movable platform raises the shuffled deck to be used by the dealer.

Reverse Engineering

IOActive reverse engineered the software running on the DM1 in an attempt to understand the operation, Random Number Generator (RNG), and entropy. The ROM code is executed on an AT89S53 (8051) CPU and was extracted from the M27C512 EEPROM.

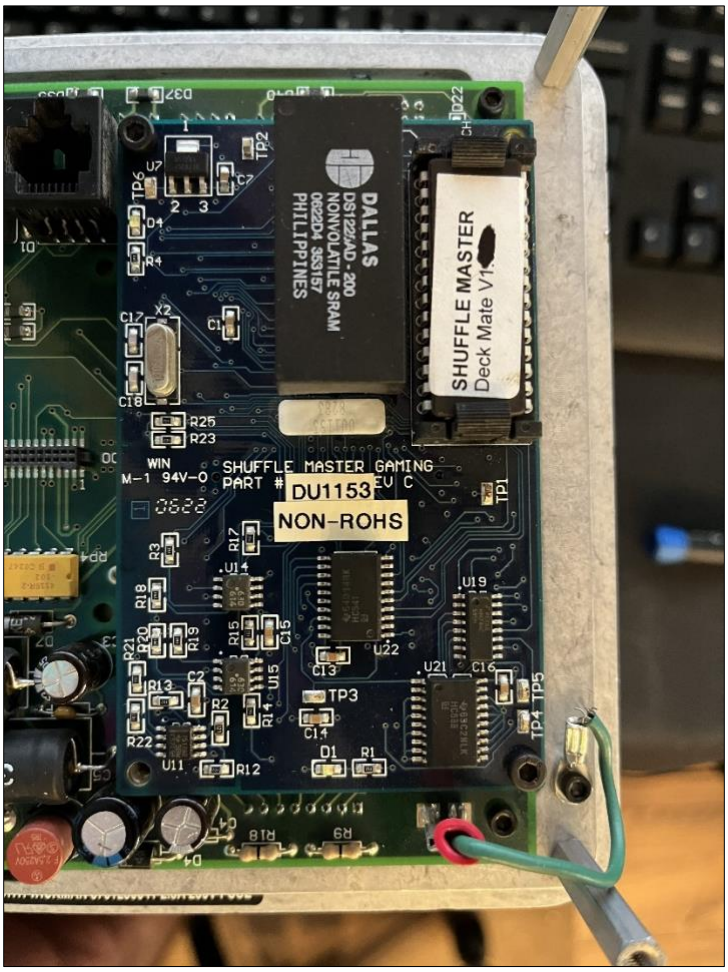


Figure 16. DM1 Machine Controller Board with EEPROM

The firmware did not include any symbols or debug information. Despite this, the reverse engineering efforts led to the successful identification of 504 functions, with 192 of them confidently named.

Identifying standard C functions out of ARM or x86 assembly is a relatively straightforward process. Intel 8051, on the other hand, is an eight-bit Harvard architecture with a reduced instruction set. While in principle the ISA is simpler to understand, mapping high-level functionality out of the compiler-generated logic is more challenging. The strings the binary had embedded to display messages over LCD and/or serial were vital for IOActive's analysis and allowed for the rapid identification of important state variables and function logic.

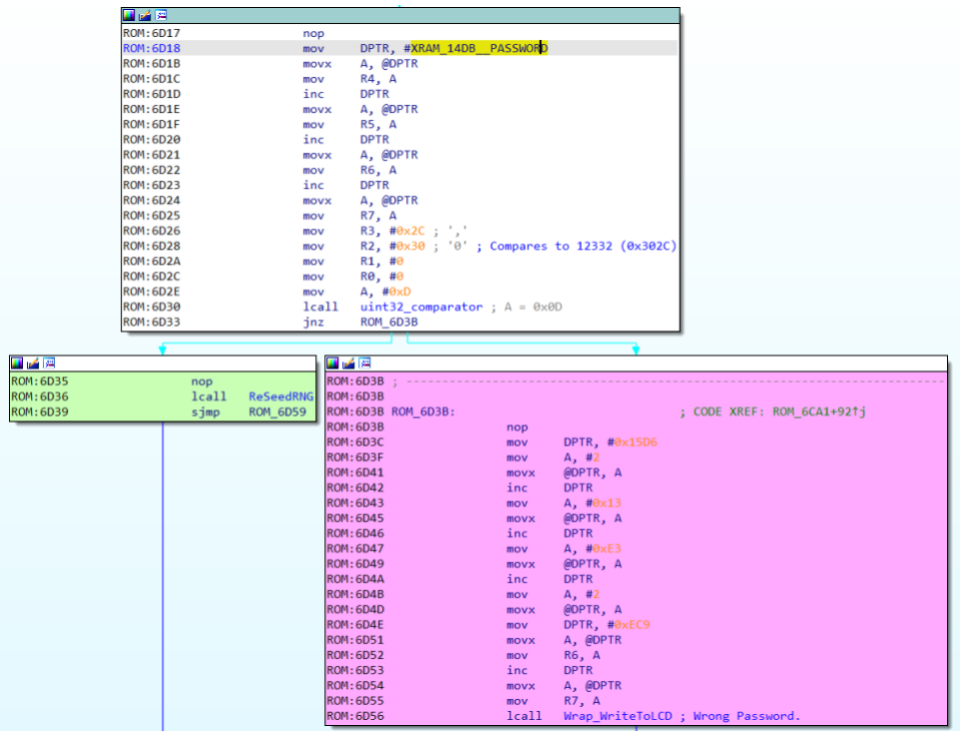


Figure 17. Function verifying password before proceeding to re-seed the RNG

Communication

The DM1 shuffler requires connections to multiple switches, sensors, and motors. Reverse engineering revealed the following relations for the 8051 MCU's ports:

- P1.4 → SPI SS
- P1.5 → MOSI
- P1.6 → MISO
- P1.7 → SCK
- P3.2 → Platform Sensor
- P3.3 → Door Home Sensor
- P3.5 → LCD State

The designers chose the Serial Peripheral Interface (SPI) protocol to communicate with the rest of the components:

- Platform Motor: moves the platform up and down
- Feeder Motor: feeds cards to the shuffling compartment
- Gripper Motor: grips cards
- Speed-Up Motor: controls the speed the card rolls into the shuffling compartment
- LCD Screen
- Buttons

- EEPROM (to store and retrieve configuration settings)

Entropy and Random Number Generator

The DM1 shuffler allows operators to "re-seed" the device to provide new entropy for randomness. The AT89S53 includes a timer that supports various modes. The shuffler will initialize Timer0 to Mode 1 (16-bit mode) with TH0 and TL0 registers connected in cascade, which it will interrupt every ~245 microseconds. IOActive's understanding is based on the following calculations:

```
The Shuffler Xtal (external crystal oscillator) is 11.0592 MHz
The core needs 12 clock periods per machine cycle.

The Timer Clock rate is then:
XtalFreq / 12 => 11059200 / 12 = 921600 = 921.6 KHz

This means the TH0|TL0 counter is increased by 1 every 1.085
microseconds.

Because TH0|TL0 is set to 0xFF1E, there are 226 increments before an
overflow occurs.Timer 0 overflow sets TF0 flag generating an interrupt
request.

As a result, there is an interrupt every 1.08506 * 226 = 245.22
microseconds.
```

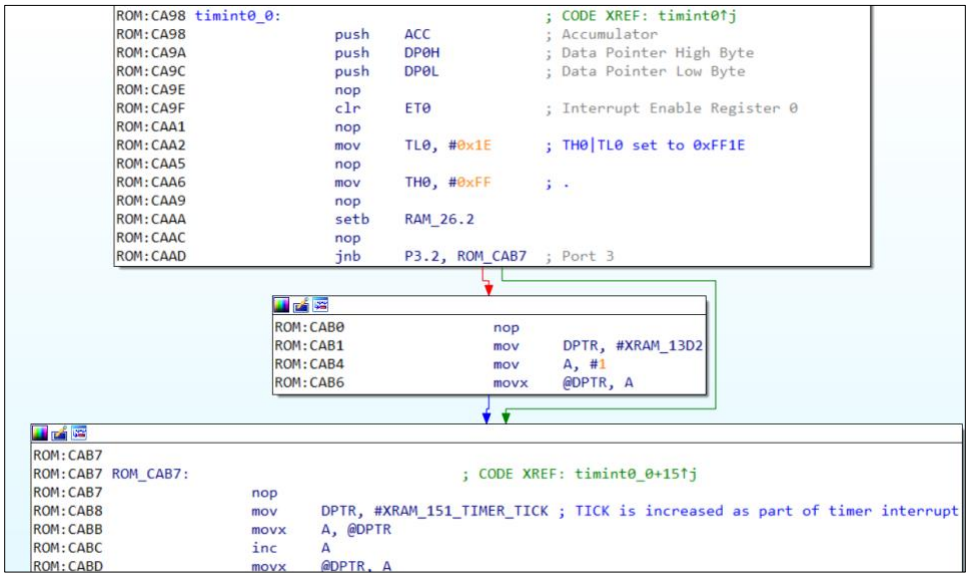


Figure 18. Timer-0 Interrupt Handler

When choosing to re-seed the device, the operator will be prompted to press the green button on the device four times sequentially and the device will use the configured timer to source the entropy. Here is an implementation of the function:

```
reseed_rng() {
   UINT32 *seed = XRAM_014Dh;
   *seed = 0;
   for (int i = 0; i < 4; i++ ) {
      // Wait for green button input
      BYTE timer_count = XRAM_151_TimerTick;
      *seed = *seed | (((UINT32) timer_count) << 8 * i);
   }
}
```

The RNG is the following common Linear Congruential Generator (LCG):

```
Seed = 0x19660d * seed + 0x3c6ef35f
```

Shuffling

The state machine for a shuffle cycle is very complex, as it must coordinate all of the hardware parts involved in the process and check for errors at each step. From a logical standpoint, however, it can be reduced to the following steps:

1. Cards are physically loaded into the first compartment.
2. Based on the configured game settings, the algorithm expects a specific number of cards. For poker, this number is 52.
3. A new deck configuration is randomly generated. This is represented by an array of numbered positions which later indicates how many cards the set of grippers should grip at each step.
4. Shuffling starts, and the deck configuration is "executed."
5. Upon error-free completion, the shuffled deck becomes available.

See DM1 Shuffling Algorithm in C in Appendix A for a C-code representation of the exact algorithm the DM1 follows to generate randomized decks and calculate how many cards need to be gripped.

```
[n3k@thanatos shuffler]$ ./GenerateDeck
Randomized Deck:
34 48 25 37 49 43 11 24 28 01 12 16 19 51 10 21 31 45 27 38 08 33 20 14 09 07
46 04 35 32 00 26 06 23 41 18 39 03 13 44 30 15 17 42 02 22 40 47 05 50 36 29
Grips to perform:
00 01 00 02 04 03 00 01 03 00 02 03 04 13 01 06 10 14 09 14 01 13 07 05 02 01
23 01 20 18 00 16 03 15 28 12 29 02 11 34 24 13 15 36 02 21 37 44 05 48 35 29

[n3k@thanatos shuffler]$
```

Figure 19. DM1 Shuffling Algorithm Execution

Figure 19 shows an example of a deck randomization and the associated card grips that need to occur. The values of the randomized deck indicate the position where the current bottom card needs to be. Considering the first five iterations step-by-step (italicized are gripped cards, underlined are the new sorted card):

1. The first card needs to be in position 34. No grip needs to happen because there are no other cards yet. The current shuffling compartment looks like [34].

2. The second card needs to be in position 48, which means this time a grip is necessary. The compartment now looks like [*34*, 48].
3. The third card needs to be in position 25, no grip is necessary: [25, 34, 48].
4. The fourth card needs to be in position 37, which means two cards need to be gripped: [*25*, *34*, 37, 48].
5. The fifth card needs to be in position 49, meaning four cards need to be gripped: [*25*, *34*, *37*, *48*, 49].
6. The process repeats until all cards are placed into their proper location.

Cheating Techniques for Poker (Proof of Concept)

Due to the limitations in the hardware architecture of the DM1, if a bad actor has internal access to the device, they can flash or replace the EEPROM chip and the MCU will simply execute the code. AT89S53 MCUs are considered old technology and lack the concept of secure boot. An attacker can backdoor the firmware and cheat with the aid of a DM1. In addition, the DM1 does not support GAT, so there is no trivial way for an auditor to detect if the EEPROM on the device is running unmodified code and would require physically dumping the EEPROM to compare. This section provides three examples of firmware modifications that aid in cheating.

Bypassing Card Count Detections

The DM1 keeps track of the number of cards that were fed into the shuffling compartment. This permits the detection of missing or extra cards. By manipulating the firmware, an attacker can alter the code logic to avoid failing when too few or too many cards are processed. This would allow an attacker at the poker table to keep an ace back (hidden up their sleeve) and the dealer would shuffle a deck of only 51 cards without being alerted.

Partial Deck Order Knowledge

Due to the way the shuffling mechanism works, where the positions of the cards are decided before a shuffle, a device with modified firmware could help a malicious dealer place cards into a known final order. For example, the code could be tweaked so that if the dealer presses the green button N times before inserting the deck to shuffle, it will use the newly implemented cheating logic. Then, this new logic would take the bottom card and place it on the top of the deck, so the malicious dealer could specifically give a certain player an ace. This could be expanded to support more cards. For example, if the dealer places two aces at the bottom of the deck an attacker could ensure that those two would be dealt to the small blind (first position). The deck configuration shown in Figure 20 achieves this behavior (assuming the dealer performs a false cut, or the game is not cutting the deck).

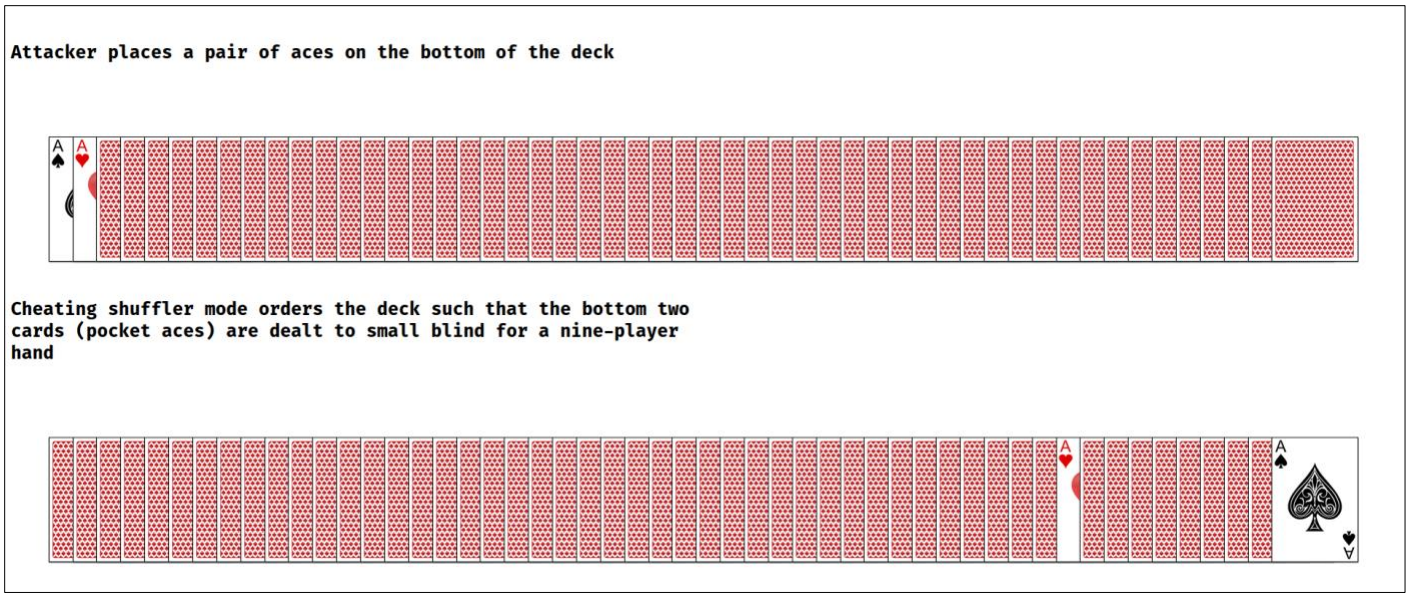


Figure 20. Dealing Aces

Similar code could be hardcoded, and even if the attacker was not conspiring with the dealer, they could peek at the bottom card when the dealer inserts the deck into the shuffler. The code could then ensure that the bottom card is on top and, upon the dealer cutting the deck, the attackers knows that card is essentially out of play.

False Shuffle

With modified firmware, the device could be configured to perform false shuffles periodically (every other shuffle, every N shuffles, etc.) or after a rogue dealer presses a button sequence before the shuffle. The dealer can keep the deck in the same state as after the previous hand and the cheater will be aware of the previous flop, turn, and river cards, as well as their hand, which would be on the top of the deck. Given this knowledge, upon the deck being cut, those known cards could be considered dead, giving the cheater a significant edge in calculating outs (cards left in the deck that can make a good hand) for the current hand.

Deck Mate 2

The DM2 is the most recent ShuffleMaster device and is used in poker rooms around the world, including the World Series of Poker (WSOP). It improves on its predecessor in multiple areas with a 22-second deck shuffle, on-board card recognition, built-in timer for a player action clock, and the ability to sort the deck.

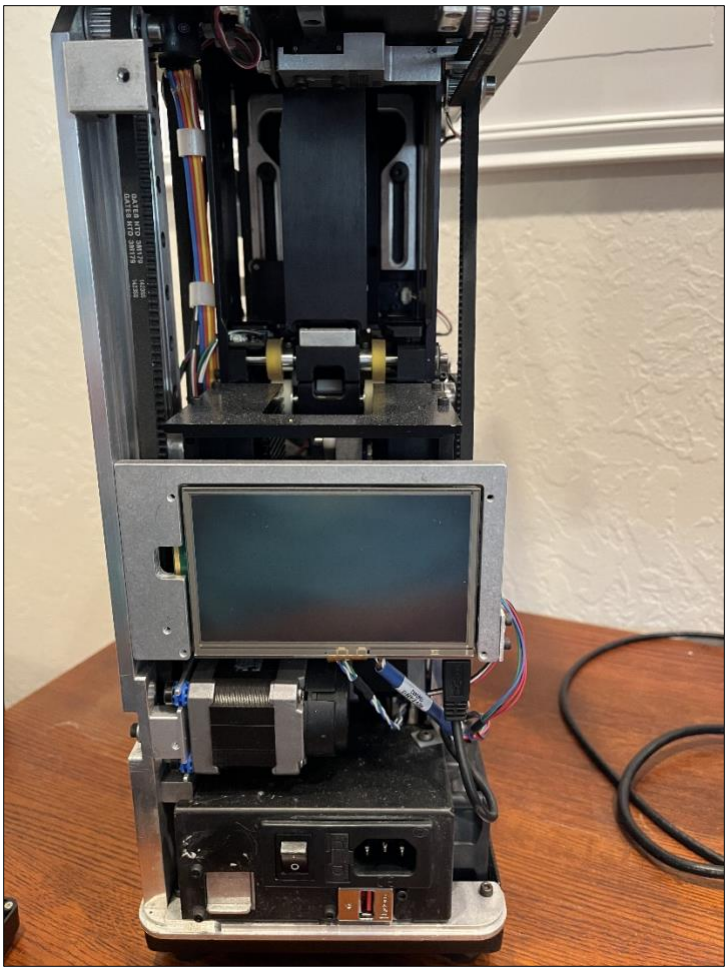


Figure 21. DM2 Front Internals

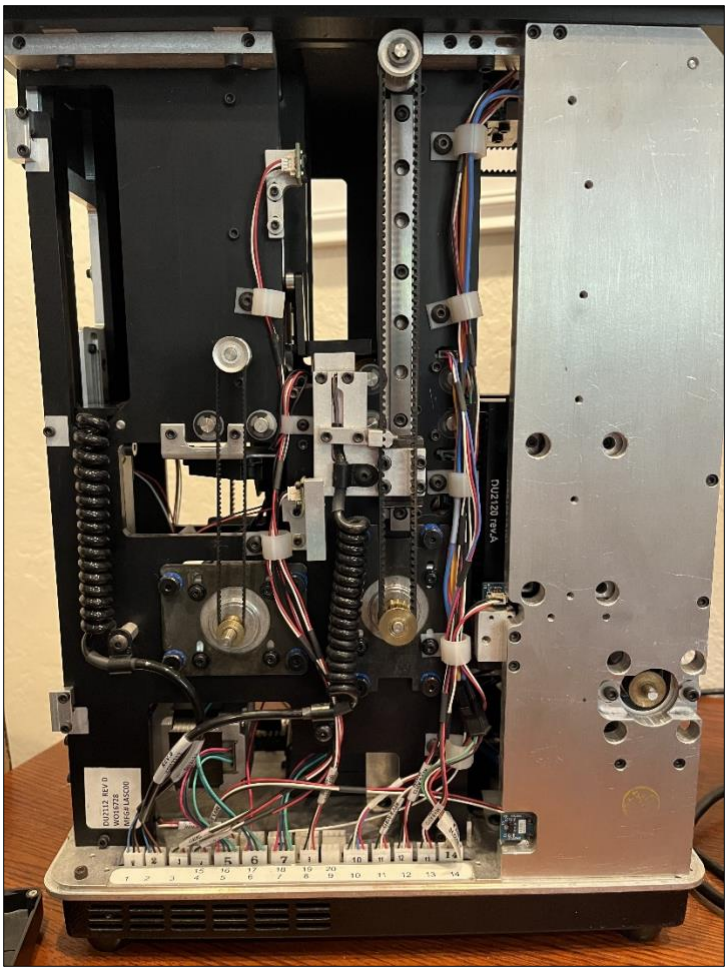


Figure 22. DM2 Left-side Internals

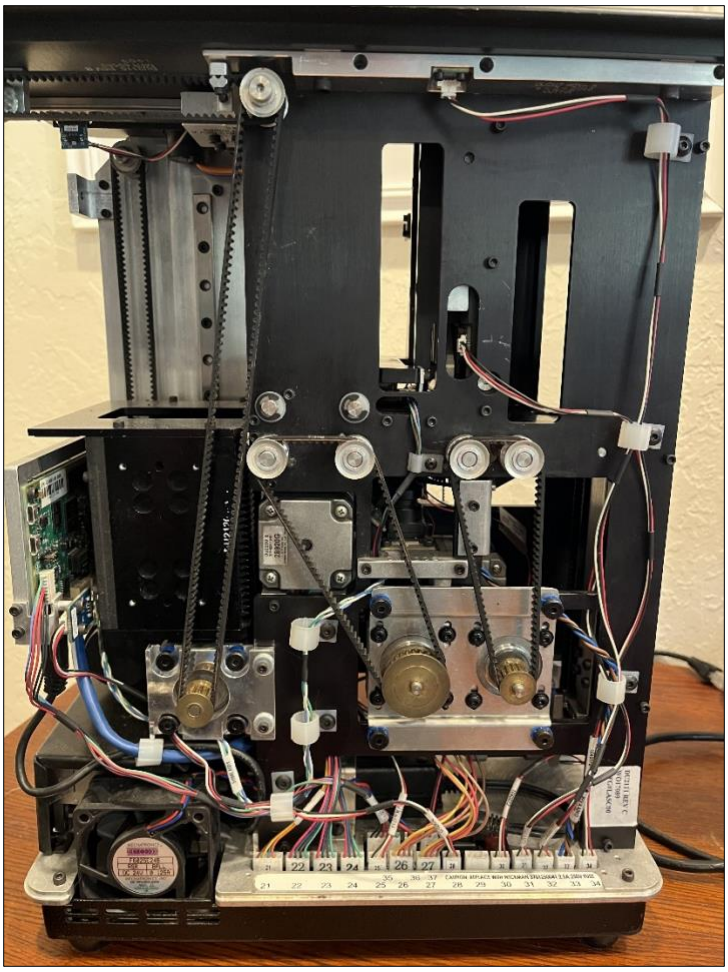


Figure 23. DM2 Right-side Internals

Physical Shuffling Mechanism

The DM2 contains a card rack which has 27 slots and can hold up to 54 cards. The slot location where each card will be placed is decided via the RNG. The shuffler processes each card one at a time, reading the card with the camera and noting the rank and suit of the card, then deciding its slot location. If the card sorting feature is being used, the slot will be selected to order the cards by their rank and suit. During this process, the shuffler will keep track of each rank and suit that has been processed, as well as an overall count. At the end of shuffling, if a discrepancy is detected, the dealer will be alerted via the external display module and a failed shuffler notice displaying the discrepancy, whether it is extra or missing cards.

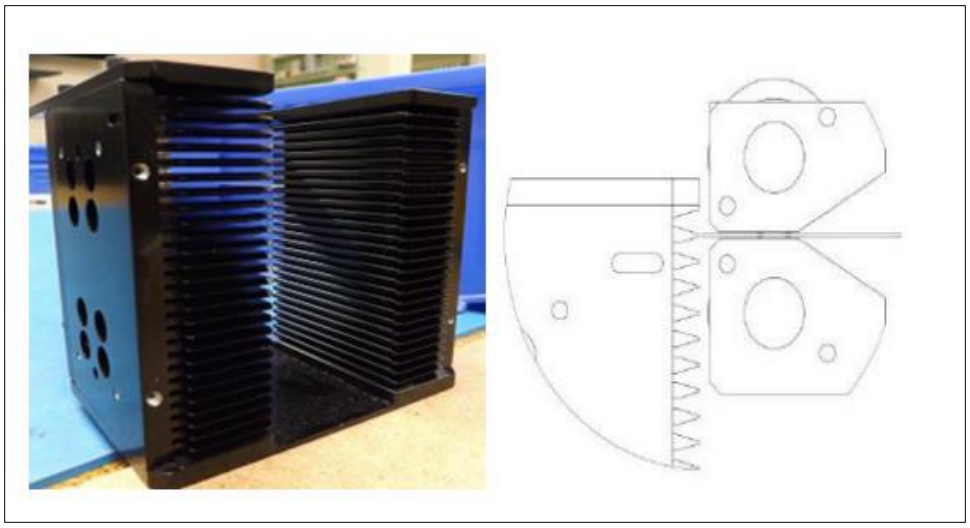


Figure 24. Scientific Games DM2 Participant Edition

Architecture

The DM2 hardware consists of a machine controller board and two display module boards.

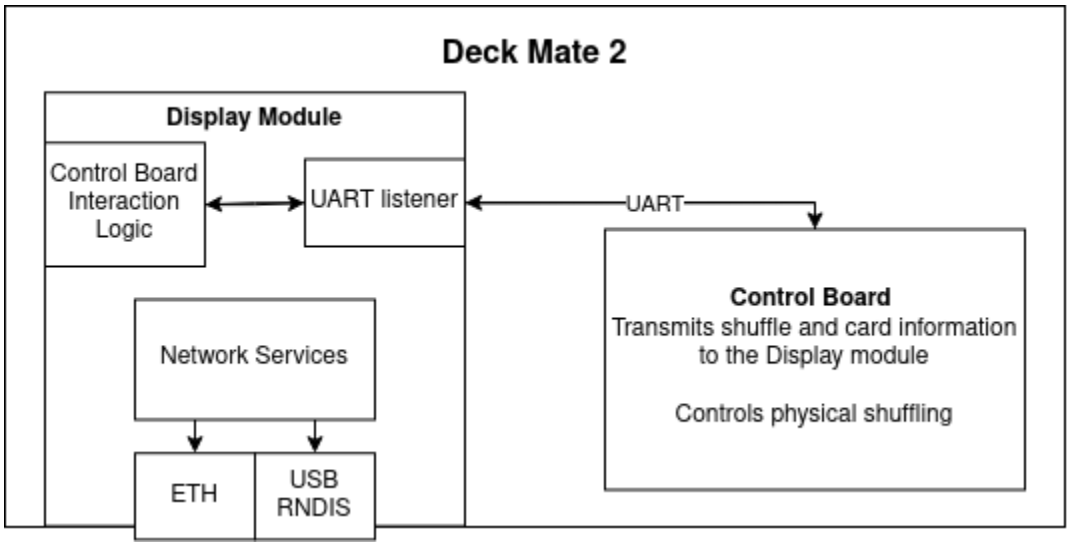


Figure 25. Deck Mate 2 Architecture Diagram

Machine Controller Board

The machine controller board consists of an NXP LPC1769 ARM CPU and a LATTICE LCMX02 1200HC FPGA.

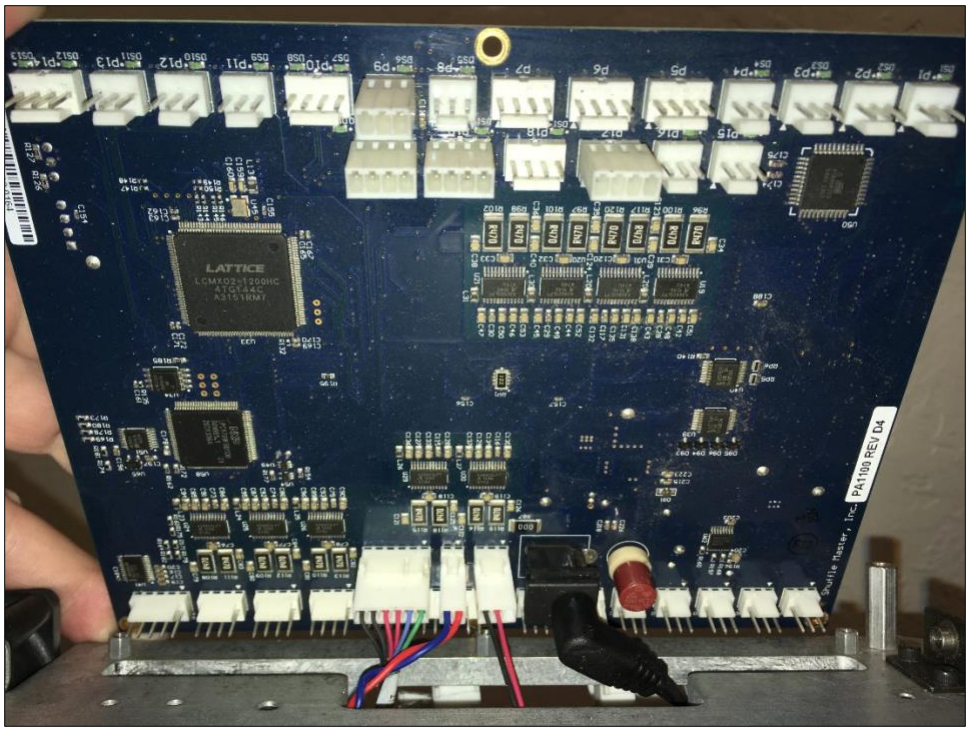


Figure 26. DM2 Machine Controller Board Top

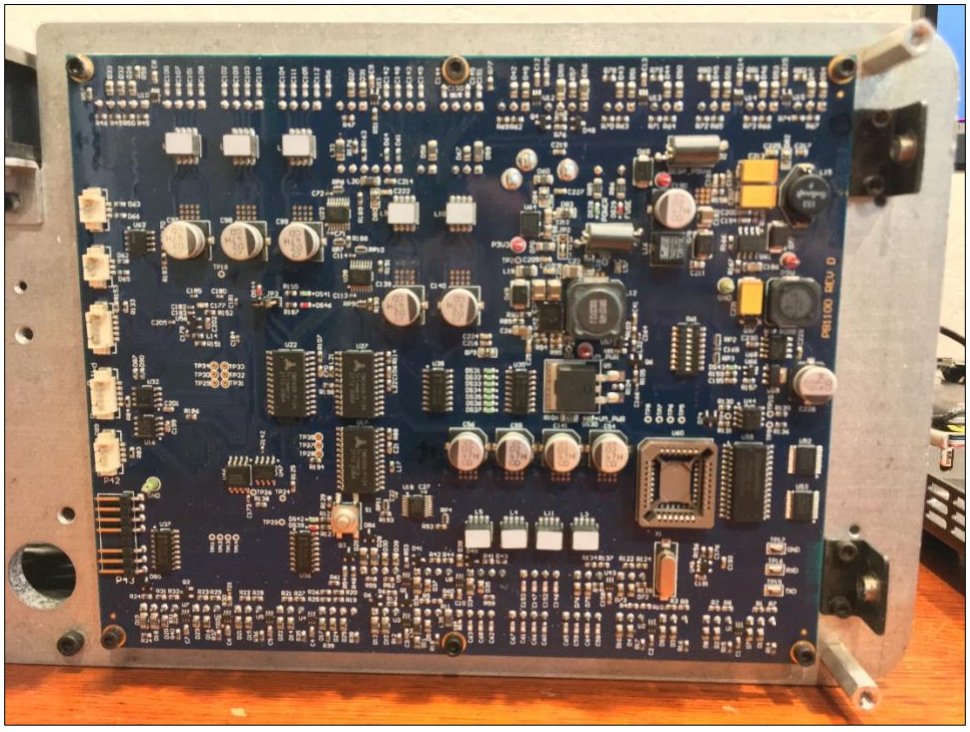


Figure 27. DM2 Machine Controller Board Bottom

The LATTICE FPGA is used for Optical Character Recognition (OCR) to read the card value and suit and then communicate these details over a serial interface to the NXP software. The on-board recognition feature is used to alert dealers when an abnormality has been detected while shuffling a deck, this could be the lack of

a card or the introduction of a duplicate. If a cheater were to replace a card in their hand with an ace of spades, which they had hidden up their sleeve, the next shuffle would detect that a second ace of spades is present in the deck, halt shuffling, and alert the dealer of which card was discovered to have a duplicate.

The NXP ARM CPU runs the main shuffler firmware code which operates all motor, sensor, and switch functions. The software is written with Quantum Leap's QP/C Real-Time Operating System (RTOS). The software consists of several 'active objects,' which are event-driven, execute under their own threads and communicate with other objects asynchronously via events. This design approach turns the shuffler into essentially a giant state machine.

The primary function of the machine control board is to shuffle a deck of cards. It is responsible for actuating all mechanical components (servos for card placement, motors, etc.), as well as receiving the data from the cameras and acting on that data as described above. It is also responsible for responding to requests from the display module and pushing notifications to the display module if configured, which takes place over the UART connection between the two boards.

Software Architecture

As described above, the shuffler controller software is built on top of QP/C, an event-driven RTOS. Thus, it is broken up into various active objects, each of which are designated for the handling of specific events. These events include external input (i.e. via UART, from the FPGA/cameras, etc.) or internal signals, such as those triggered by timers or faults.

The firmware did not include symbols or any debug information. In order to understand the firmware structure and context, IOActive referenced the open-source QP/C OS code and documentation, which made it possible to identify common patterns within code written for QP/C, as well as to quickly identify important functions such as object initialization and entry points. IOActive also leveraged the presence of debug strings found within various functions, which gave hints as to a given function's role within the greater state machine. A total of 974 functions were defined within the firmware, with IOActive able to confidently identify and name 566 of them via reverse engineering.

IOActive was able to analyze the standard structure of various binaries using the framework and how the compiler tended to organize the code, which greatly helped in understanding where logic existed in the DM2 NXP firmware. IOActive was able to define initialization steps for the device, as well as ultimately tracking down the full state machine flow including the following state objects:

- Calibrate
- CardWeight
- Controller
- Diagnostic
- FeedElev
- Homing
- JamRecover
- Packer
- PickOff
- Platform

- Rack
- ShuffleCards
- SpeedUp
- UI
- Unloader
- Utility

IOActive focused on the objects associated with the positioning of cards during a shuffle, entropy acquisition and RNG for shuffle placement, the processing of cards identified by the camera, and logging functionality, owing to their potential relevance for cheating with the shuffler.

At a high level, a shuffle is initiated as described by the flow chart in Figure 28, which involves interactions between several active objects moving through several states, communicating via events.

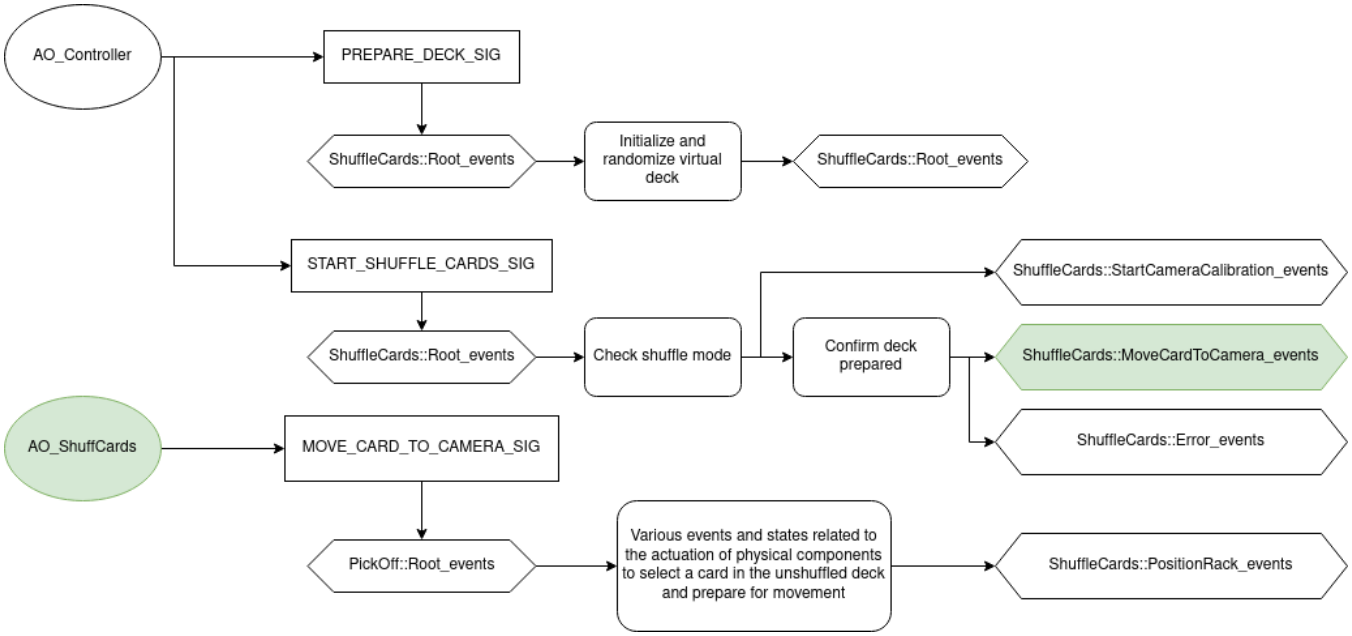


Figure 28. Shuffle Initiation State Diagram

During the physical shuffling of cards, once the deck order has been decided by the process above, the process of placing each card involves the interactions demonstrated in Figure 29.

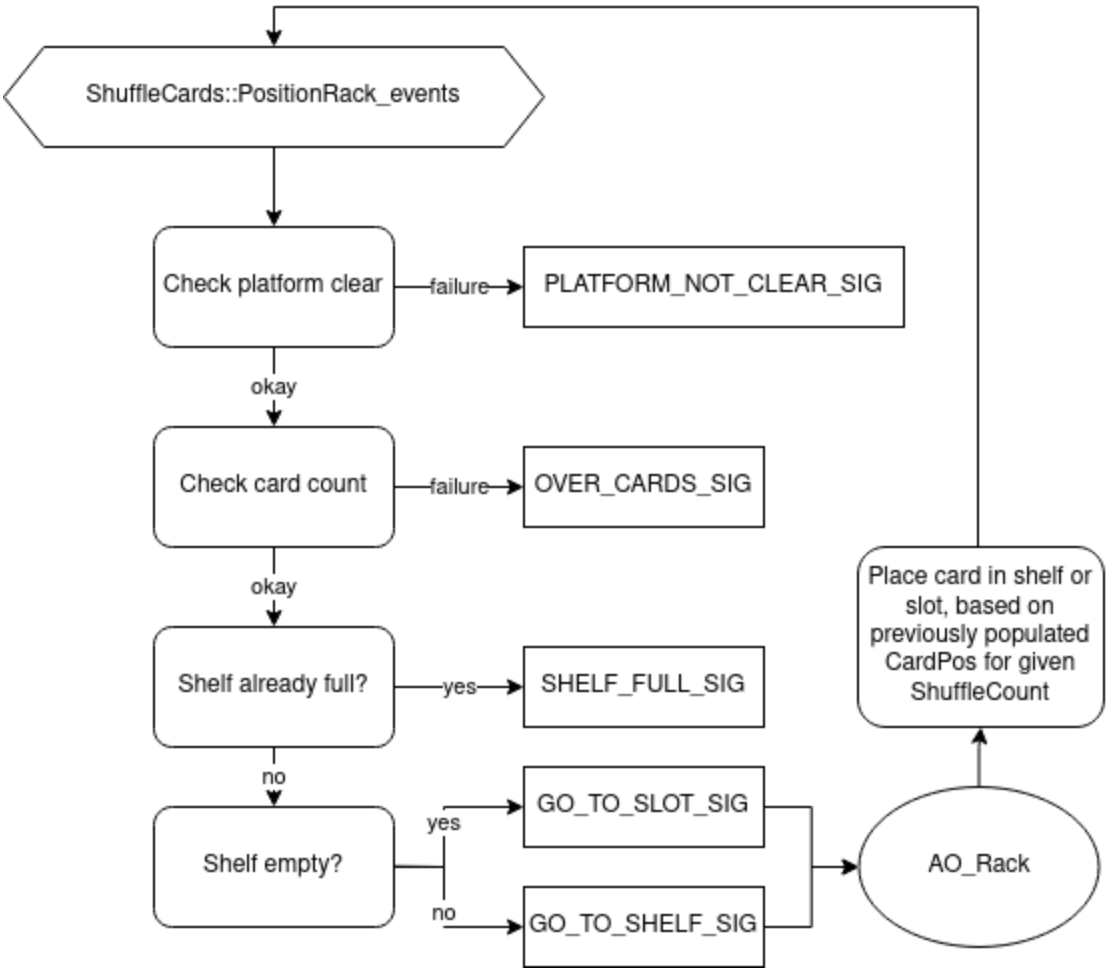


Figure 29. Physical Shuffling Logic Per-Card

Entropy and Random Number Generator

The DM2 machine controller board does not feature any hardware or cryptographically secure RNG. Rather, it relies on software pseudo-RNG (PRNG), where a relatively unpredictable seed value is used to deterministically generate pseudo-random data.

The RNG is initialized as part of system init, in a function dubbed *initialize_system_features* by IOActive.

```
0001b04c  int32_t initialize_system_features()

0001b050      CPU_clock_configuration_mb()
0001b054      initialize_SSP0_mb()
0001b058      initialize_SSP1_mb()
0001b05c      configure_pins_and_perform_ssp_transmission_and_enable_interrupt()
0001b068      initialize_ADC_mb(0xf4240)
0001b06c      initialize_RIT_repetitive_interrupt_timer_mb()
0001b070      configure_gpio_pins()
0001b074      initialize_uarts_and_various_objects()
0001b080      initialize_device_info_struct(device_info: &device_info)
0001b0a4      initialize_RNG(rng: &RNG, print_description: print_description, get_RITIMER_COUNTER: get_RITIMER_COUNTER, &device_info)
0001b0a8      initialize_items_mb()
0001b0ac      sub_1b0cc()
0001b0b0      LinuxComInitialize()
0001b0b4      sub_2d984()
0001b0c8      return print_description(9, "NXP 1.0.172")
```

Figure 30. initialize_system_features

RNG initialization includes setting function pointers for logging and a function which retrieves the current value of onboard Repetitive Interrupt Timer (*RITIMER*). This function will be called later, during the seeding process.

```
0003b08c  int32_t initialize_RNG(struct rng_struct* rng, int32_t (& logger)(int32_t arg1, uint8_t* arg2), uint32_t (& get_RITIMER_COUNTER)(), int32_t arg4)

0003b09e      rng->print_description = logger
0003b0a4      rng->get_RITIMER_COUNTER_fp = get_RITIMER_COUNTER
0003b0d2      initialize_named_global_object(obj: &rng->seed_obj, name: "RNGseed", seed: 0, 0xffffffff, 0, 1, arg4)
0003b0e6      rng->seed = get_item_value(&rng->seed_obj)
0003b0ee      if (rng->seed != 0) {
0003b10e          initialize_seed(rng)
0003b10e      } else {
0003b0f6          rng->seed_status = UNITITIALIZED
0003b0fa          rng->print_description(0xa, "Init: RNG requires seeding")
0003b104      }
0003b11e      return 1
```

Figure 31. initialize_RNG

The seed is initialized to zero, as seen in Figure 31. In this case, a flag is set to 2 (determined by IOActive to indicate a fully uninitialized seed), and a log indicates seeding is required. Another function, dubbed *SeedRNG*, subsequently checks this flag at the top of the function. If it is set, a seed is generated as shown in Figure 32.

```
0003b424  void SeedRNG(struct rng_struct* rng)

0003b432      if (rng->seed_status != SEEDED) {
0003b43a          if (rng->seed_status != IN_PROGRESS) {
0003b44c              RITimer_Counter = rng->get_RITIMER_COUNTER_fp()
0003b454              rng->seed_status = IN_PROGRESS
0003b454          } else {
0003b462              int32_t current_timer_value = rng->get_RITIMER_COUNTER_fp()
0003b478              set_item_value(internal_item_obj: &rng->seed_obj, RITimer_Counter * current_timer_value)
0003b48c              rng->seed = get_item_value(&rng->seed_obj)
0003b494              if (rng->seed != 0) {
0003b4ac                  rng->print_description(0xb, "SeedRNG: seed complete")
0003b4c2                  rng->seed_status = SEEDED
0003b4c2              } else {
0003b498                  rng->print_description(0xc, "SeedRNG: unable to update seed")
0003b4a2              }
0003b4a2          }
0003b4a2      }
```

Figure 32. SeedRNG

If the seed state is UNINITIALIZED, the current value of the *RITIMER* is queried, and a global variable is set to this value. The seed state is updated to *IN_PROGRESS*, and the function returns. *SeedRNG* will be called again (since the seed state is still not *SEEDED*), and when this occurs, the lower portion of the function will execute. Here, the *RITIMER* is queried again, and this is multiplied with the previous *RITIMER* value stored in the global variable to generate the initial seed.

Based on this review, IOActive found that this technique is likely to produce a sufficiently unpredictable initial seed value. Due to these two timer values being queried in separate function calls, and owing to the event-based real-time nature of QP/C, it appears that the time elapsed between the two calls of this function cannot be deterministically predicted, and thus the delta between the two timer samples will also not be predictable under normal operating conditions.

The *RITIMER* by default counts at a rate equal to the system clock rate, a maximum of 120MHz in the case of the NXP LPC1769. This means that timing differences on the scale of tens of nanoseconds will alter the sampled counter value, and thus the final multiplied seed value. Therefore, it is relatively unlikely that an attacker could accurately guess the seed, even assuming they knew when the machine was powered on, and the time elapsed between reset and the initial calling of the *SeedRNG* function.

During the randomization functions, the device will use a common LCG for the RNG:

```
0x19660d * seed + 0x3c6ef35f
```

Shuffling

The *ShuffleCards* active object is responsible for shuffling the playing cards. The main event handler for this object accepts the following events:

```
RNG_SIMULATION_MODE
PREPARE_DECK
START_SHUFFLE_CARDS
```

Deck Preparation

The actual shuffling of the cards happens prior to any movement of the physical deck of cards. That is, the position of each card is determined in software, and the determined ordered is then later applied to the physical deck. The function used to determine the order of the shuffled deck was dubbed *RandomizeCards*, and was reimplemented by IOActive to make the logic clearer, as seen in Figure 33.

```
int
RandomizeCards(struct RNG *rng, short total_cards,
               short top_pos, short bottom_pos,
               short *target_positions, short rng_buf_size)
{

    // populate ordered list of card positions, 0
    // to 51 in normal case
    for (int i = 0; i < total_cards; i++) {
        rng->ordered_positions[i] = i + bottom_pos;
    }

    short max_slot = top_pos - bottom_pos;
    int read_position = 0;

    for (; read_position < total_cards - 1; read_position++, max_slot--) {

        // generate random position, then set the target positions at the current
        // offset into the shuffled deck to the card position found in the ordered
        // cards array at the random position
        int rand_slot = generate_random_int_in_range(rng, max_slot, 0);
        target_positions[read_position] = rng->ordered_positions[rand_slot];

        int curr_slot = rand_slot;
        int next_slot = rand_slot + 1;

        // remove the already used position from the ordered positions array
        // so it can't be reused, allowing for duplicate rand_slots
        // to not produce duplicate positions
        for (; next_slot < max_slot; curr_slot++, next_slot++) {
            rng->ordered_positions[curr_slot] = rng->ordered_positions[next_slot];
        }
    }
    // put the last remaining position in the ordered positions array into the final slot
    // of the shuffled cards. This is fine and still not predictable, since
    // the fact that this position was not selected was a result of calls to generate
    // random function happening to miss it.
    target_positions[read_position] = rng->ordered_positions[0];
    return 1;
}
```

Figure 33. Function to Determine Deck Order

Essentially, the *generate_random_int_in_range* function uses the previously discussed RNG to generate a random integer in the supplied range. This value represents a target position in the shuffled deck. The chosen value is then inserted into the *target_positions* array at the current sequential position, which starts at 0 and ends at 51, in the case of a standard 52 card deck.

The *target_positions* argument to the function is then used in subsequent logic, after this function is called. For example, the *ShuffleCards::PositionRack* event handler, which reads positions from this array before positioning the rack so that a card can be inserted into a slot on the shuffled side of the device internals.

It is worth noting that the shuffler supports several modes, including standard shuffle (with no modifiers and card placement) and sort (where cards are sorted by suit). These are selected by the shuffler operator via the device's UI. There are various checks in the randomization and shuffling logic which check for the currently selected mode and will alter behavior accordingly.

The above-described logic could be altered or patched in order to adjust the shuffling behavior and cheat. For example, an additional mode could be introduced that would put the cards in a predetermined order and covertly triggered by an attacker.

Communications

The NXP firmware communicates with the Linux environment over UART. There is a serial protocol which has various "items" which can be queried and set. For example, the following is what happens over UART when the device product service starts and authenticates the NXP firmware and initializes:

It is possible for an attacker connected to this bus to control various configuration options and components of the device by issuing these commands. For example, opening the top and preparing to receive a deck.

Display Module Boards

The display module boards have two separate configurations: internal and external. Within the DM2, an internal display module board is mounted and available for technicians working on the machine when the case is removed. The external display module board is connected to the DM2 over the USB port and generally mounted to the poker table and available for the dealer to use during play. The software allows for changing game configurations, starting the clock (player action timer), and sorting the deck upon completion of the game.

```
set.request name=UCommand value=3,36
{fpga}w a=03 d=04
{fpga}r a=01
{fpga}w a=03 d=00
{fpga}w a=03 d=04
{fpga}w a=70 d=c0
{fpga}w a=70 d=80
{fpga}w a=71 d=19
{fpga}w a=71 d=00
{fpga}w a=71 d=00
{fpga}w a=71 d=00
{fpga}r a=73
{fpga}r a=73
{fpga}r a=73
{fpga}r a=73
{fpga}r a=73
{fpga}r a=73
```

```
{fpga}r a=73
{fpga}r a=73
{fpga}w a=f0 d=80
{fpga}w a=03 d=00
{fpga}w a=03 d=04
{fpga}r a=01
{fpga}w a=03 d=00
{fpga}w a=03 d=04
{fpga}r a=01
{fpga}w a=03 d=00
{fpga}w a=03 d=04
{fpga}r a=00
{fpga}w a=03 d=00
{fpga}w a=03 d=04
{fpga}r a=02
{fpga}w a=03 d=00
set.request name=UCommand value=2,42
set.request name=UCommand value=3,36
notify.request name=* enable=1
notify.request name=MInputSensors enable=0
set.request name=UDisableButtonProcessing value=1
get.request name=MSoftwareVersion
get.request name=MProductName
set.request name=UNumberOfDecks value=1
set.request name=UCardsPerShuffle value=52
set.request name=UCardsPerDeck value=52
set.request name=UCardWidth value=58
set.request name=UCardRecEnabled value=1
get.request name=MInteractiveStatus
set.request name=USortOrder value=0
get.request name=MHardCounter
set.request name=UGameNumber value=0
notify.request name=MInPusherActualPosition enable=0
notify.request name=MInPusherOffset enable=0
notify.request name=MOutPusherOffset enable=0
notify.request name=MMagazineZero enable=0
notify.request name=MMagazineActualPosition enable=0
notify.request name=MGapOpenerActualPostion enable=0
notify.request name=MGapActualPosition enable=0
set.request name=UScreenID value=Init.Init
get.request name=MAuthentic
get.request name=MInteractiveProcess
set.request name=UCommand value=1,25
set.request name=UScreenID value=DeckMate2.Welcome
get.request name=MInteractiveStatus
set.request name=UCommand value=2,30,1
get.request name=MCardWidth
set.request name=UCardWidth value=58
set.request name=UCommand value=3,37
```

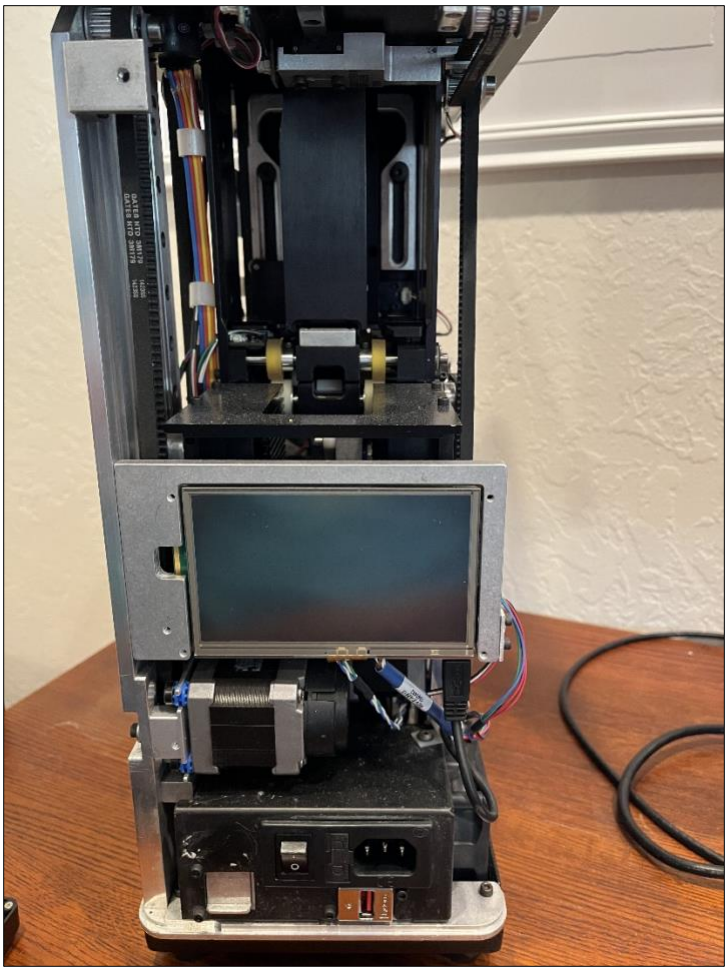


Figure 34. DM2 Internal Display Module

Figure 35. DM2 External Display Module

The display module board runs the i.MX28 NXP ARM CPU and is, in fact, the Reach Technology touchscreen display development module.

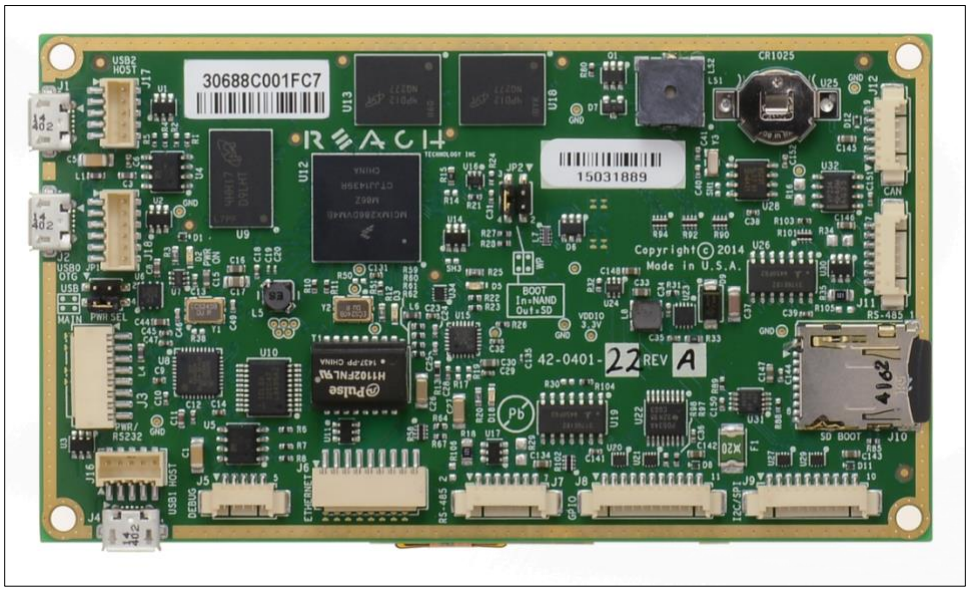


Figure 36. Reach Technology Touchscreen Display Development Module

The board runs Linux 2.6.35.3 and communicates with the machine controller board over UART serial communication. The device supports booting off NAND memory or via an SD card. This allowed IOActive to simply create a Linux bootable environment on an SD card, boot it, and dump the NAND memory for analysis. The board has multiple interfaces, including an inverted UART interface running 115200 8n1 5v.

The system is configured with a single root user. Due to the low complexity of the root user's password, if the password hash were obtained by an attacker, it can quickly be cracked by basic consumer hardware. Once the password is compromised, it appears to be statically deployed on all display module boards, giving an attacker the ability to log in to any DM2 shuffler via local UART shell or over SSH.

The device exposes a number of services via the Ethernet port exposed on the back of the DM2.

```
PORT     STATE SERVICE
22/tcp    open ssh
23/tcp    open telnet
80/tcp    open http
139/tcp   open netbios-ssn
445/tcp   open microsoft-ds
6000/tcp  open X11
```

The HTTP web server, which is running lighttpd and PHP, lists details about the device and allows for configuration changes, such as setting the date/time, name, and deck options. In addition, there are four different web authentication accounts which grant various access.

Table 4. Web authentication accounts

Password (redacted)	Description
*****	Basic access: Shuffler label, time, deck library, history and restart.
********	Basic access + network configuration options.
*****	Basic access + network + Card ID Deck calibration, images, error details and restore default settings.
******	All previous access + Advanced Options and Advanced Calibration of motors, cameras and sensors.

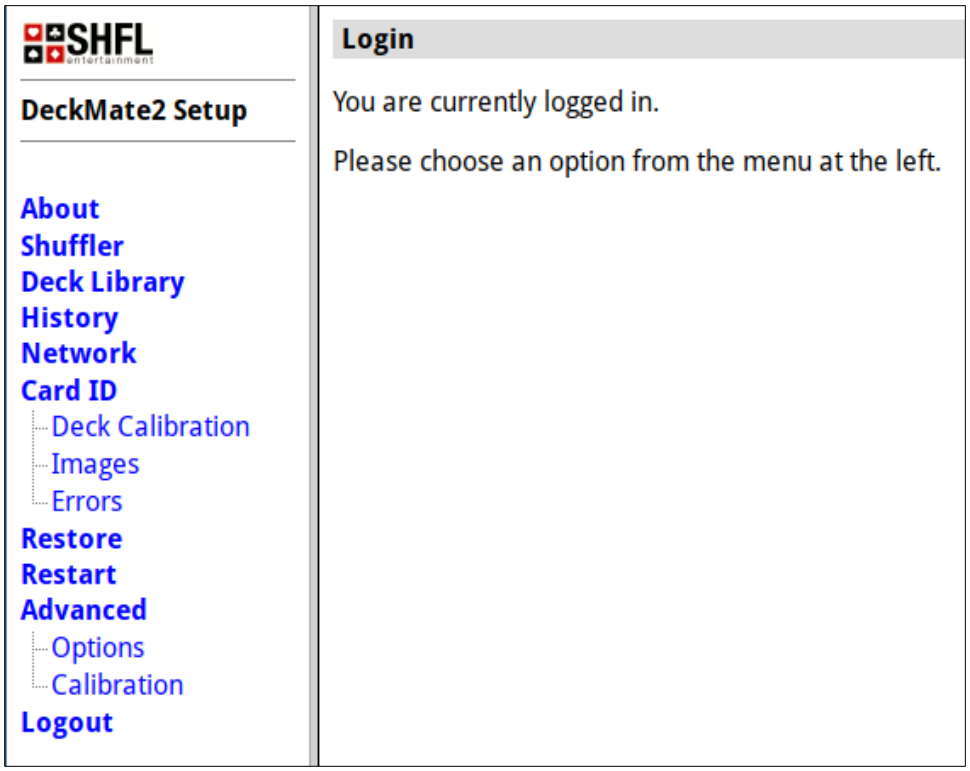


Figure 37. DM2 Authenticated Web Interface

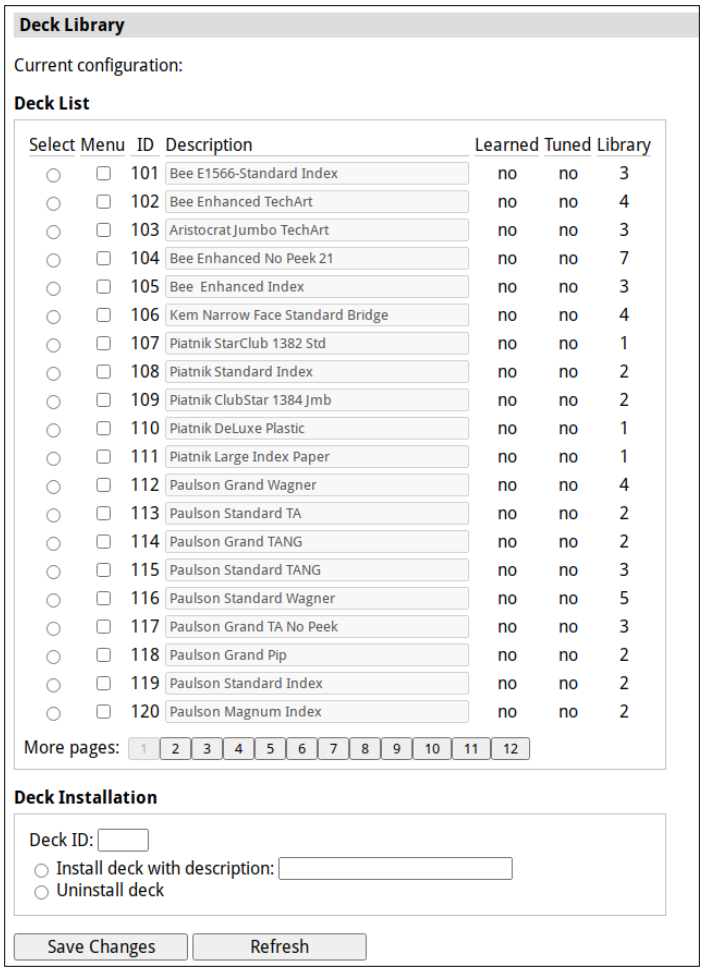


Figure 38. DM2 Web Interface Deck Library Configuration

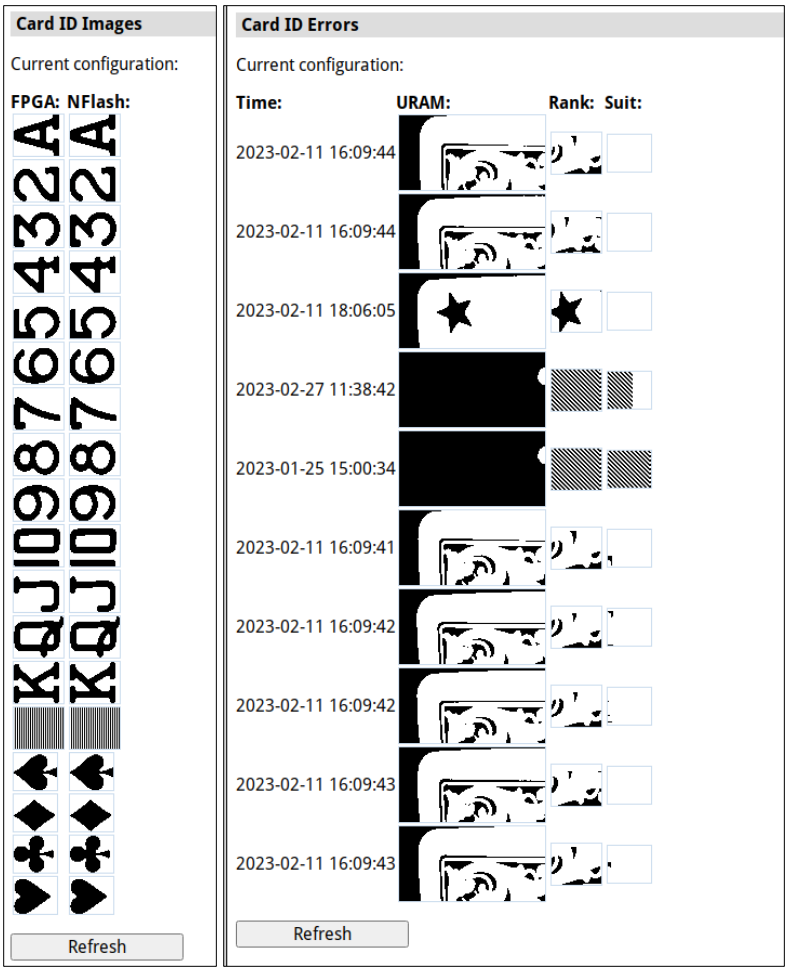


Figure 39. DM2 Web Interface Configured Card Images and Error reports

These web server passwords are static and hardcoded in the binary logic of the services. If the passwords were ever compromised by a bad actor, they could be used to authenticate to the web application of any exposed DM2 shuffler.

The SMB feature does not appear to be expected for use by customers and it was not immediately clear why the service is exposed.

The X11 feature is used specifically when the display module board is connected over USB. It will be referred to as the remote display, and it is technically running as an RNDIS Ethernet gadget. The main internal display module monitors the USB port, and upon detection of a remote display, it will then communicate via Ethernet over USB over port 6000 using X11 networking. The device will SSH to the main device and then configure itself to display the X11 software window on its own display screen to the dealer.

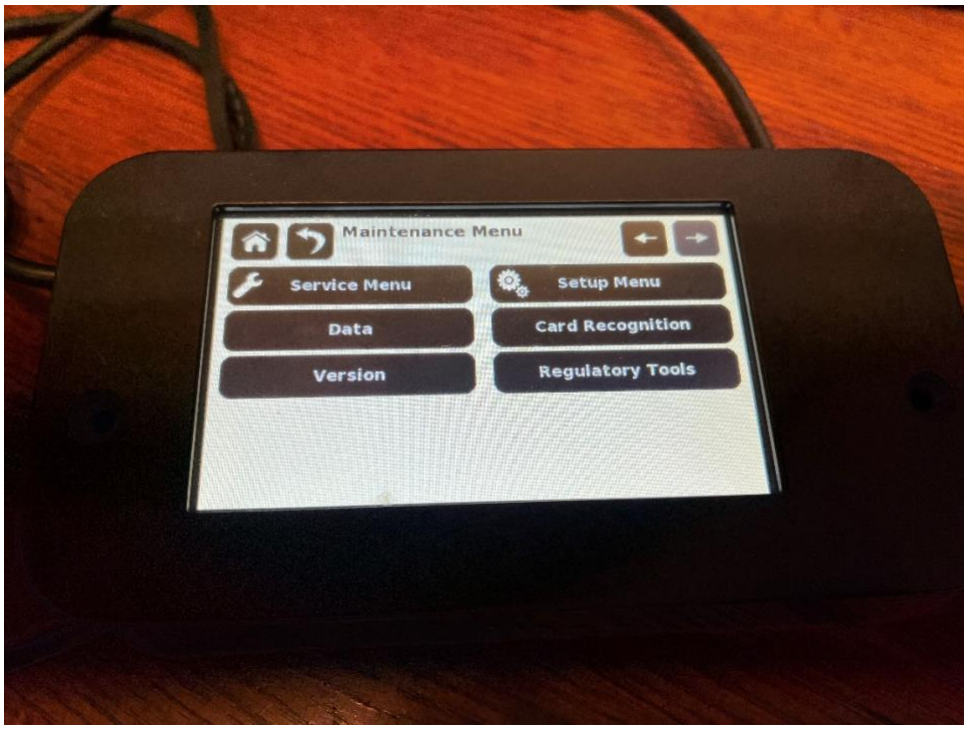


Figure 40. DM2 External Display Module Maintenance Menu via X11 forwarding

The SSH service unnecessarily exposes itself externally which increases attack surface and allows attackers to remotely connect to the device using compromised static credentials. The Telnet service does not appear to be used by internal software.

Secure boot is not implemented on the device, meaning an attacker with means to access the eMMC (physical or otherwise) could modify the bootloader, kernel, or data on the disk without being detected. An attacker can use this to compromise the device by running altered malicious code.

The device also does not employ any mechanism to verify that the contents of the filesystems have not been altered prior to mounting them. Consider the case of a device that has been actively compromised, where an attacker has gained shell access. In an ideal security situation, this initial access should not provide the attacker with any means of persistent access or the ability to make changes to the device; however, in the case of the DM2, the filesystems are neither immutable nor validated at boot. This means that once access is gained, an attacker can make arbitrary changes to the software that resides on these filesystems. This includes device functionality, including software as well as configuration.

As an example, IOActive was able to make patch changes to various service and firmware binaries on disk to alter the behavior of the device and make further analysis easier. Similarly, these changes can be made via any other means of accessing the filesystems of the device. For example, physical access to the eMMC on the board allows for similar attacks, since the contents of the eMMC are not encrypted. All of these changes were persistent between boots and at no point did the device attempt to cryptographically validate the contents of the software running on the device against some known, trusted state.

Software Updates

The DM2 allows for software updates to be installed via a USB flash drive. The software update files are tar-compressed, self-extracting bash scripts that are encrypted into chunks and packaged within a custom TLV container format.

The updates will eventually unpack into a directory with a setup script, configuration files, and the binaries that are installed. For example, an NXP software package (ARM Shuffler Firmware) installation directory structure looks like this:

```
.
├── install
│   └── DeckMate2_NXP
├── nxp
│   ├── binary
│   │   └── DeckMate2_NXP_1.0.118.bin.gz
│   ├── nxp
│   ├── nxp-flash
│   └── uninstall-nxp
├── setup
├── setup.conf
└── sys_profile
    └── syscontroller-nxp.sh
```

Each package container has three types of chunks:

Table 5. Package container contents

Type	Description
0	Version
1	IV Chunk
2	Data Chunk
3	Digest Chunk

As the container format is processed, each data chunk is decrypted using AES 128 CFB mode with a 16-byte key and 16-byte IV and the SHA1 context is updated. Ultimately, the SHA1-HMAC digest is generated using the 16-byte encryption key as the HMAC and is used for validation when decrypting. This SHA1-HMAC implementation was found to have a vulnerability when updating the final digest, which increases the likelihood of an attacker generating an update file with a SHA1 collision. This could lead to the DM2 attempting to decrypt and process/install an attacker-controlled update file. Due to the nature of the DM2 using self-extracting shell scripts, the attacker would simply need the data to decrypt into junk data in which the beginning of that junk data contains valid shell commands that aid in device compromise.

IOActive found that the decryption utility was incorrectly performing the HMAC steps of the SHA1-HMAC process. When attempting to update the final digest with the previously generated digest over the update data, the program will unintentionally only use the first four bytes of the digest instead of the entire 16-byte value.

```
00009098  ldr     r3, [r11,  #-0x10] {var_14}
0000909c  add     r3, r3, #0x190 {key_obj::sha1_info}
000090a0  mov     r0, r3  // key_obj->sha1_info
000090a4  ldr     r1, [r11,  #-0x14] {var_18}  // digest buffer
000090a8  mov     r2, #0x4  // length (only first 4 bytes)
000090ac  bl      sha1_update_mb
000090b0  ldr     r3, [r11,  #-0x10] {var_14}
000090b4  add     r3, r3, #0x190 {key_obj::sha1_info}
000090b8  mov     r0, r3
000090bc  ldr     r1, [r11,  #-0x14] {var_18}
000090c0  bl      sha1_final_mb
000090c4  sub     sp, r11, #0xc
000090c8  ldm     sp, {r11, sp, pc} {__saved_r11} {__saved_r11} {var_8}
```

Figure 41. Incorrect HMAC Steps

Presumably there is a bug in the code when providing the length value for `sha1_update()`; it may be returning the `sizeof()` a pointer (four bytes) instead of the size of the buffer (16 bytes). This increases the likelihood of an attacker generating an SHA1 collision due to the HMAC only using four bytes.

The following is an example of a decrypted software update showing the self-extracting script which is appended before the tar compressed archive. DM2 Update Decryption Utility in Appendix A includes an update decryption utility written in C.

```
#
# product = DeckMate2
# component = CardRec
# name = DeckMate2_CardRec
# version = 5.0.023
# date = 2013-02-27 12:00:00
# company = Shuffle Master Inc.
# system = SysController
# machine = armv5tejl
# kernel = Linux
# os = GNU/Linux
#

if (echo "$0" | /bin/grep -s -q "^-"); then
  echo
  echo "This program must be executed with sh."
  echo
  return 1
fi
command="$@"
installer_dir='DeckMate2_CardRec_5.0.023'
target_dir='/tmp'
linenum=$(/usr/bin/awk "/^---$/ { print NR + 1; exit 0; }" "$0")
/bin/rm -rf "$target_dir/$installer_dir"
/usr/bin/tail -n+$linenum "$0" | /bin/tar xz -C "$target_dir"
status="0"
if [ "$command" == "" ]; then
  echo
  echo "The files have been extracted into a new directory"
```

```
  echo "at \"$target_dir/$installer_dir\""
  echo
  echo "Run setup in the new directory."
  echo
else
  "$target_dir/$installer_dir/setup" "$@"
  status="$?"
  /bin/rm -rf "$target_dir/$installer_dir"
fi
exit "$status"
---
# tar archive data here #
```

Due to the nature of the self-extracting shell scripts, this method would allow the attacker to obtain arbitrary command and code execution as the root user. In addition, the 16-byte encryption key appears to be static across all DM2 shufflers, so if an attacker compromises a shuffler and exposes the encryption key, it is possible for the attacker to encrypt a modified software update and provide it on a USB drive.

IOActive observed that the DM2 software updates lack cryptographic signature verification. This would allow an attacker to trivially encrypt modified software packages that unpack and execute on the device. An attacker could introduce a malicious backdoor into the software that gives them the ability to inspect or alter the state and behavior of the shuffler.

Cheating Techniques for Poker (Proofs of Concept)

Deck Order Monitoring

One potential method to game an advantage in a game of poker, leverages the lack of firmware update security on the NXP machine controller board, the lack of adequate security for authentication to the display module board, and the inadequate physical security of the shuffler unit.

Overview

The stock firmware running on the NXP control board is, internally, *aware* of the order of cards after a shuffle has taken place. Additionally, during operation, it does log (over UART connection to the display module board) some information about the status of the shuffle; however, these logs did not contain the order of the cards; in order to force the NXP board to transmit this information, a binary patch was applied to allow for this information to be transmitted.

Initial compromise of the device takes place via a hardware implant, in the simplest case a device inserted into the USB port exposed on the back of the shuffler. This USB port allows for Ethernet-over-USB communications (as this is its intended purpose, normally utilized by gaming technicians for maintenance or remote display for dealer interaction/configuration), and once inserted will allow for network communication with the device. It should also be noted that the shuffler also features a real Ethernet port. Should the device be networked using this port, this attack could similarly be performed over said network.

Once networked with the machine, the attacking device can connect to the internal display board Linux environment using the weak, hardcoded root credentials which all shufflers are configured with. With shell access, it is possible to fully control the display module board.

NXP Firmware Modification

All communication between the display module board and machine controller board takes place over a UART connection. In addition to operational communications, this UART connection also provides access to the machine controller board's bootloader/firmware update mechanism. A binary utility for reflashing the machine controller board was present in the display module board's filesystem, known as *nxp_flash*.

While integrity checking (via SHA-1 hash) is implemented for NXP firmware images, no cryptographic verification of these images is implemented, meaning flashing of the patched firmware binary only requires an update of the SHA-1 hash after the patch has been applied. There is a mechanism by which gaming operators can verify that a software or firmware entity has not been modified vis-à-vis the GAT protocol; however, it is possible to falsify the data returned by the GAT service to mimic the original firmware.

This can be done by modifying the service that implements the GAT feature such that instead of communicating directly with the firmware chip for firmware verification, it reports the original hash by performing the digest checks against the original firmware binary stored on disk. Even if this was not possible, attackers could still bypass these GAT checks by keeping a dictionary of original bytes that were modified during patching and update the authentication mechanism to reference these original bytes in order to always ensure the unmodified digests are returned. This is due to the inherent design flaws described in Game Authentication Terminal (GAT) regarding requesting a compromised device to inform you if it is indeed compromised.

Thus, it is possible not only to reflash the machine controller board with firmware patched to allow for cheating, but also to disguise this update in a way that would avoid detection by the standard verification tools. Of course, byte-by-byte comparison between a legitimate firmware image and the modified one would reveal the changes, so this technique would not stand up to highly technical investigation of the targeted shuffler.

Shuffler-side Data Processing

Once the machine controller board has been modified to report card order information, a UART listener is set up on the display module board to consume the incoming data. This listener reads all the card order data as it comes in over UART and forwards it over a network socket to the attached implant (either local via USB or over the network), which parses the raw UART data and extracts from it the actual order of the cards.

```
Shuffler connected - receiving cards
ShuffleCount:  0 Card Position: 15 :: Rank: 4, Suit: C, Sequence:  1
ShuffleCount:  1 Card Position: 23 :: Rank: 6, Suit: S, Sequence:  2
ShuffleCount:  2 Card Position: 20 :: Rank: 2, Suit: S, Sequence:  3
ShuffleCount:  3 Card Position: 28 :: Rank: 9, Suit: D, Sequence:  4
ShuffleCount:  4 Card Position: 33 :: Rank: 8, Suit: H, Sequence:  5
ShuffleCount:  5 Card Position: 25 :: Rank: Q, Suit: S, Sequence:  6
ShuffleCount:  6 Card Position: 26 :: Rank: K, Suit: S, Sequence:  7
ShuffleCount:  7 Card Position:  4 :: Rank: 9, Suit: H, Sequence:  8
ShuffleCount:  8 Card Position: 11 :: Rank: 2, Suit: H, Sequence:  9
ShuffleCount:  9 Card Position:  0 :: Rank: 6, Suit: D, Sequence: 10
ShuffleCount: 10 Card Position: 48 :: Rank: 5, Suit: D, Sequence: 11
ShuffleCount: 11 Card Position:  5 :: Rank: K, Suit: H, Sequence: 12
ShuffleCount: 12 Card Position: 43 :: Rank: Q, Suit: D, Sequence: 13
ShuffleCount: 13 Card Position: 45 :: Rank: 6, Suit: H, Sequence: 14
ShuffleCount: 14 Card Position: 34 :: Rank: 7, Suit: C, Sequence: 15
ShuffleCount: 15 Card Position: 38 :: Rank: T, Suit: D, Sequence: 16
ShuffleCount: 16 Card Position: 12 :: Rank: 9, Suit: S, Sequence: 17
ShuffleCount: 17 Card Position: 49 :: Rank: 8, Suit: D, Sequence: 18
ShuffleCount: 18 Card Position: 21 :: Rank: 4, Suit: S, Sequence: 19
ShuffleCount: 19 Card Position: 40 :: Rank: A, Suit: D, Sequence: 20
ShuffleCount: 20 Card Position: 24 :: Rank: 4, Suit: H, Sequence: 21
ShuffleCount: 21 Card Position:  2 :: Rank: T, Suit: C, Sequence: 22
ShuffleCount: 22 Card Position: 19 :: Rank: 3, Suit: C, Sequence: 23
ShuffleCount: 23 Card Position: 22 :: Rank: K, Suit: C, Sequence: 24
ShuffleCount: 24 Card Position:  1 :: Rank: 2, Suit: C, Sequence: 25
ShuffleCount: 25 Card Position: 52 :: Rank: T, Suit: H, Sequence: 26
ShuffleCount: 26 Card Position: 51 :: Rank: 3, Suit: H, Sequence: 27
ShuffleCount: 27 Card Position: 37 :: Rank: 3, Suit: S, Sequence: 28
ShuffleCount: 28 Card Position: 29 :: Rank: A, Suit: H, Sequence: 29
ShuffleCount: 29 Card Position: 47 :: Rank: J, Suit: C, Sequence: 30
ShuffleCount: 30 Card Position: 50 :: Rank: 4, Suit: D, Sequence: 31
ShuffleCount: 31 Card Position: 16 :: Rank: A, Suit: S, Sequence: 32
ShuffleCount: 32 Card Position: 35 :: Rank: Q, Suit: C, Sequence: 33
ShuffleCount: 33 Card Position: 32 :: Rank: 8, Suit: S, Sequence: 34
ShuffleCount: 34 Card Position: 44 :: Rank: K, Suit: D, Sequence: 35
ShuffleCount: 35 Card Position: 39 :: Rank: T, Suit: S, Sequence: 36
ShuffleCount: 36 Card Position: 10 :: Rank: 9, Suit: C, Sequence: 37
ShuffleCount: 37 Card Position: 30 :: Rank: 2, Suit: D, Sequence: 38
ShuffleCount: 38 Card Position: 41 :: Rank: J, Suit: D, Sequence: 39
ShuffleCount: 39 Card Position: 18 :: Rank: 5, Suit: S, Sequence: 40
ShuffleCount: 40 Card Position:  6 :: Rank: 5, Suit: C, Sequence: 41
ShuffleCount: 41 Card Position: 36 :: Rank: 7, Suit: D, Sequence: 42
ShuffleCount: 42 Card Position: 53 :: Rank: 8, Suit: C, Sequence: 43
ShuffleCount: 43 Card Position: 46 :: Rank: 6, Suit: C, Sequence: 44
ShuffleCount: 44 Card Position: 14 :: Rank: 7, Suit: H, Sequence: 45
ShuffleCount: 45 Card Position:  7 :: Rank: J, Suit: S, Sequence: 46
ShuffleCount: 46 Card Position: 42 :: Rank: J, Suit: H, Sequence: 47
ShuffleCount: 47 Card Position: 27 :: Rank: Q, Suit: H, Sequence: 48
ShuffleCount: 48 Card Position: 13 :: Rank: 5, Suit: H, Sequence: 49
ShuffleCount: 49 Card Position:  3 :: Rank: A, Suit: C, Sequence: 50
ShuffleCount: 50 Card Position: 31 :: Rank: 3, Suit: D, Sequence: 51
ShuffleCount: 51 Card Position: 17 :: Rank: 7, Suit: S, Sequence: 52

Received all 52 cards, deck order:

6D 2C TC AC 9H KH 5C JS 9C 2H 9S 5H 7H 4C AS 7S 5S 3C 2S 4S KC 6S 4H QS KS QH
9D AH 2D 3D 8S 8H 7C QC 7D 3S TD TS AD JD JH QD KD 6H 6C JC 5D 8D 4D 3H TH 8C
```

Figure 42. Card Order Data

While developing this proof of concept, a situation occasionally occurred in which a single card was missing from the log, likely due to buffering of log data from UART and over the network socket. Logic was added to the data processing code which could account for a missing card. More than one missing card is not handled, as the position of each of the two cards cannot be deduced, but such an event proved to be exceedingly rare. In that event, or in the event of any other issue upon reading card data, the listener will report an error upstream, and no deck information is returned for that shuffle.

```
ShuffleCount: 34 Card Position: 44 :: Rank: K, Suit: D
ShuffleCount: 35 Card Position: 39 :: Rank: T, Suit: S
ShuffleCount: 36 Card Position: 10 :: Rank: 9, Suit: C
ShuffleCount: 37 Card Position: 30 :: Rank: 2, Suit: D
ShuffleCount: 38 Card Position: 41 :: Rank: J, Suit: D
ShuffleCount: 39 Card Position: 18 :: Rank: 5, Suit: S
ShuffleCount: 40 Card Position:  6 :: Rank: 5, Suit: C
ShuffleCount: 41 Card Position: 36 :: Rank: 7, Suit: D
ShuffleCount: 42 Card Position: 53 :: Rank: 8, Suit: C
ShuffleCount: 43 Card Position: 46 :: Rank: 6, Suit: C
ShuffleCount: 44 Card Position: 14 :: Rank: 7, Suit: H
ShuffleCount: 45 Card Position:  7 :: Rank: J, Suit: S
ShuffleCount: 46 Card Position: 42 :: Rank: J, Suit: H
ShuffleCount: 47 Card Position: 27 :: Rank: Q, Suit: H
ShuffleCount: 48 Card Position: 13 :: Rank: 5, Suit: H
ShuffleCount: 49 Card Position:  3 :: Rank: A, Suit: C
ShuffleCount: 50 Card Position: 31 :: Rank: 3, Suit: D
ShuffleCount: 51 Card Position: 17 :: Rank: 7, Suit: S
51
Incomplete deck: expected 52, got 51
Only missing one card, deducing missing card and position
Identified missing card 6S at position 23

Received all 52 cards, deck order:

6D 2C TC AC 9H KH 5C JS 9C 2H 9S 5H 7H 4C AS 7S 5S 3C 2S 4S KC 4H QS 6S KS QH
9D AH 2D 3D 8S 8H 7C QC 7D 3S TD TS AD JD JH QD KD 6H 6C JC 5D 8D 4D 3H TH 8C
```

Figure 43. Error Reporting

With the full card order acquired, this information is then transmitted by some means (our example uses Bluetooth Low Energy (LE), but this could also be accomplished over for example an LTE connection or via more exotic methods such as infrared) to a mobile app that was designed as part of this proof of concept.

Data Communication

Bluetooth LE was chosen for communication between the shuffler implant and the mobile app, as it features encryption and authentication (preventing possible discovery of cheating-related transmissions OTA) and is widely supported, particularly both by mobile phones and by very small USB-based devices, such as the Raspberry Pi Zero or the even smaller Nordic nRF BLE dongle series.

Additionally, it would also be possible to install a USB-based implant internally, tying the implant directly to the data lines of the USB interface on the inside of the case. While this would decrease the likelihood of visual detection, since it is inside the case, it requires greater access to and time with the device and would therefore only be suitable for one of the insider attack scenarios, and not for the player-at-the-table scenario.

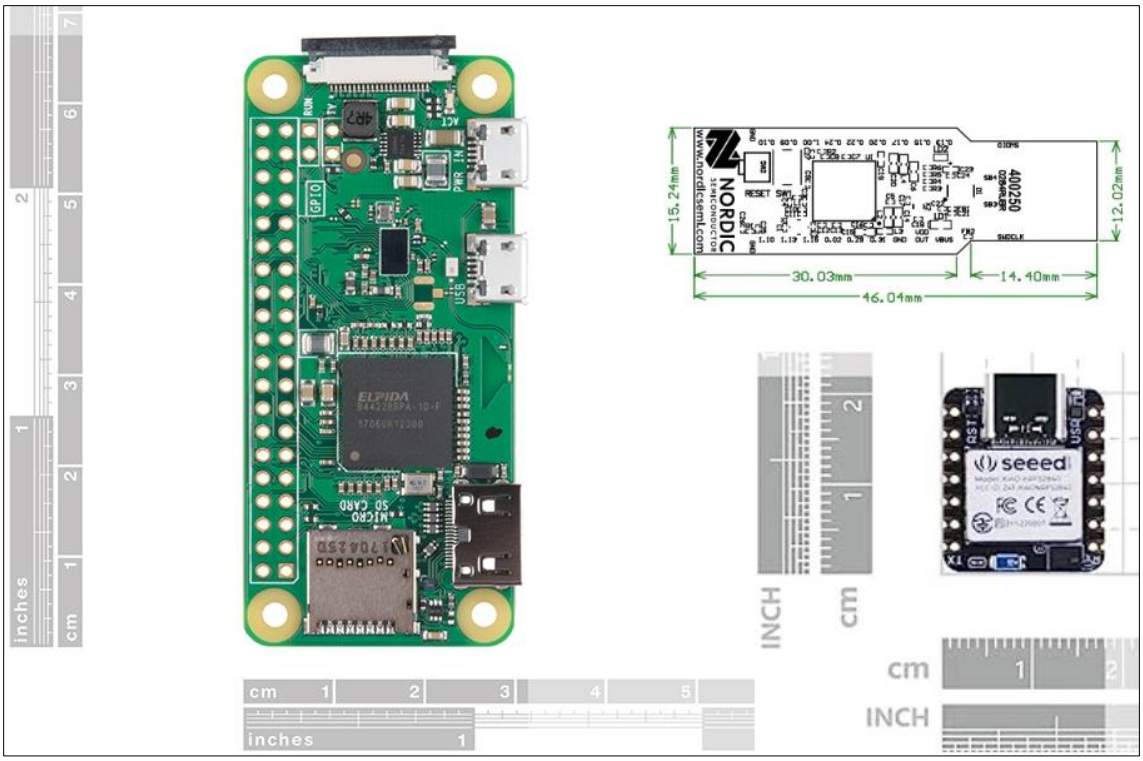


Figure 44. Raspberry Pi Zero W, Nordic nRF BLE Device and similar USB BLE Device

The architecture for communications is simple: GATT is used with a single service and single characteristic. This characteristic is read-only and offers notifications, thus allowing the mobile app to subscribe to notifications and act once the shuffler implant reports.

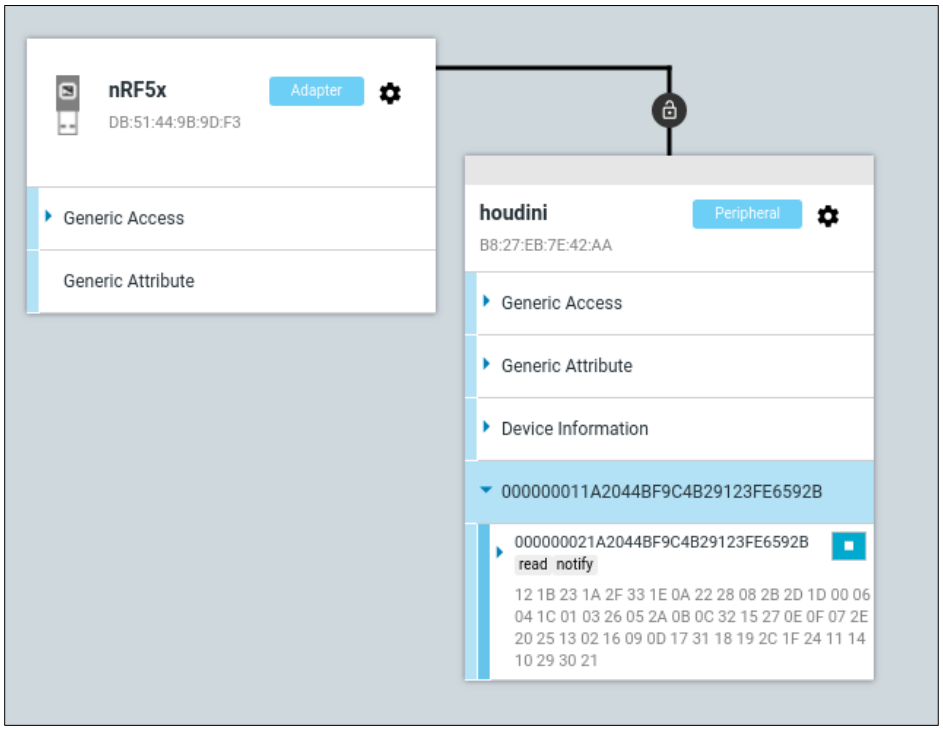


Figure 45. Communication Architecture

Mobile App

The mobile app developed for this proof of concept receives card order data over Bluetooth LE by subscribing to notifications of the card order characteristic of the shuffler implant's GATT server. Upon receipt of deck information, the user can configure various options for the current hand.

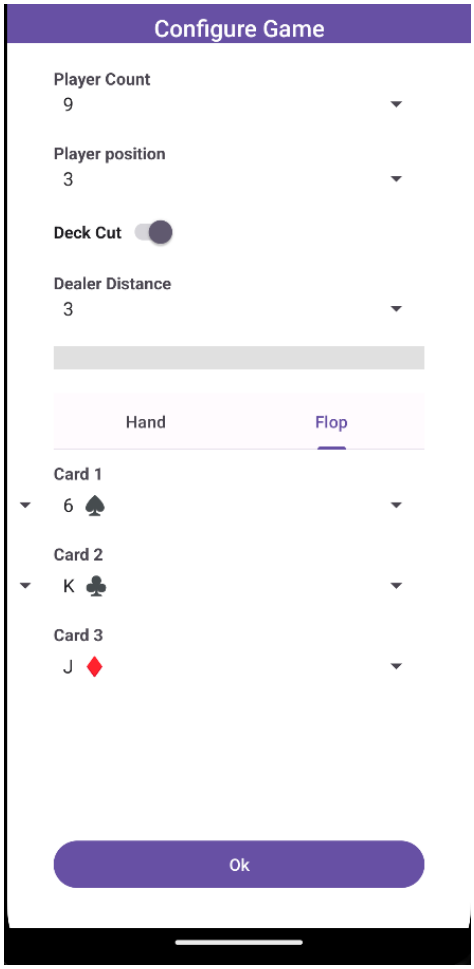


Figure 46. Mobile App

1. Player Count: The number of players in a game. This information is necessary to calculate each player's hand and the board, which will of course differ based on the number of players.
2. Player Position: The absolute seat number for the cheating player. Only used for convenience to quickly identify players by seat.
3. Deck Cut: Switch indicating whether or not the dealer cut the deck. This is discussed in more detail in Cutting the Deck.
4. Dealer Distance: The distance between the cheating player and the dealer button. Used in cut compensation logic if the cheater is using cards from hand rather than flop cards for cut calculation.
5. Hand/Flop: Card selection menu where the cards delt to the cheating player or those in the flop are supplied, for use in cut calculation.

Cutting the Deck

As discussed above, if the deck is not cut by the dealer, no additional information is necessary; the shuffler can report the final order of the cards to the mobile app, and it will be processed as such. If the deck is cut, however, additional information is required to apply the cut to the mobile app's internal deck representation. Namely, at least one card from the flop or one card from the cheater's hand (and the cheater's relative position) can be used as a marker that can be tracked between the deck order before and after the cut.

As per the 2023 WSOP official rules, to cut a deck is "to divide the deck into two face-down stacks and then reunite them by placing the bottom stack on top of the former top stack without changing the order of the cards within each stack. The cut should be approximately one half (50%) of the deck."

The phrase "without changing the order of the cards within each stack" allows the deck, at the time of the cut, to be thought of as a circular shift buffer—circular in the sense that cards moved beyond the bottom of the deck are places back at the top (and vice versa) and a shift buffer in the sense that relative to some top, or cursor, the cards may only be shifted as a whole, and not reordered.

Indexing from Flop Cards

With this in mind, and with knowledge of the deck order before the cut takes place, it follows that knowledge of the new position of a single card after the cut reveals the positions of all other cards after the cut. As stated, this is because the cards may only be *shifted* relative to the top of the deck, not reordered, and the amount one card is shifted is equal to the amount all other cards are shifted.

When a poker hand is dealt (this example uses Texas Hold 'Em rules but could be adapted for any style of poker), each player receives two cards from the top of the deck. Then, one card is "burned," and the three cards following that are the flop cards, placed on the table. Assuming knowledge of the number of players in a game, a constant number of cards are dealt prior to the flop cards, regardless of whether players fold or not. In the case of a four-player game, a total of nine cards are dealt/burned prior to the flop, making the first flop card the tenth card from the top of the deck.

Assume the first card of the flop is the two of hearts. In the post-cut deck order, the two of hearts is at index 9 (zero-indexed from the top of the deck). Assume also that the two of hearts, pre-cut as reported by the shuffler implant, was at index 25.

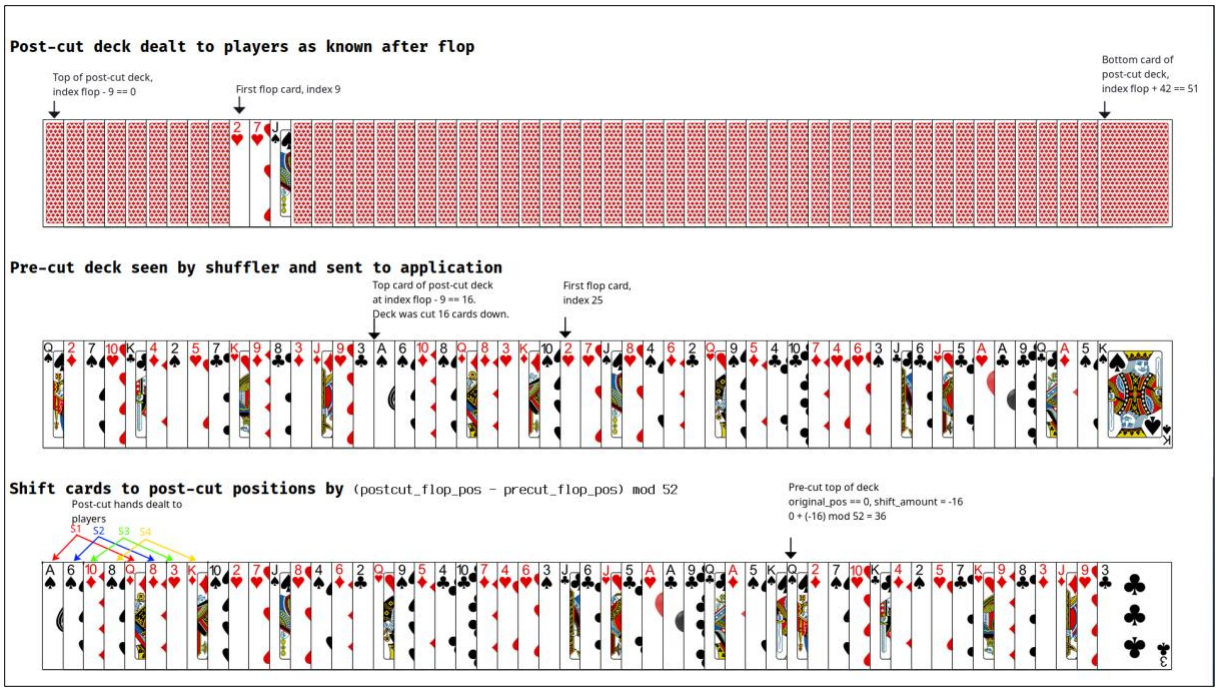


Figure 47. Deck Order Monitoring Example

As illustrated in Figure 47, the position of the cut can be calculated by subtracting the post-cut flop card position (which in this example is the predetermined index 9) from the pre-cut position of that card (in this example, 25), and taking the result mod the length of the deck/buffer, 52. In this case, 16. From here the deck can be reordered in software such that the card at the cut position is the new top of the deck, and the cards preceding it are appended to the end of the deck. At this point, the post-cut order of the cards is known, and thus the hands of all players and full flop, turn, and river are also known.

Indexing from Hand Cards

A similar technique can be performed using the first card the cheating player is dealt, rather than waiting for the flop to be revealed. In this case, the same calculation is made, but rather than a constant post-cut position as with the flop (where the position is equal to the number of players multiplied by two, plus one) the player indicates to the application their position relative to the dealer button, and this is used as the post-cut position for the card in question. From there, the process of resolving the cut is identical.

Results

The architecture diagram in Figure 48 describes the full interaction between the modified software on the DM2, the USB implant, and the mobile application.

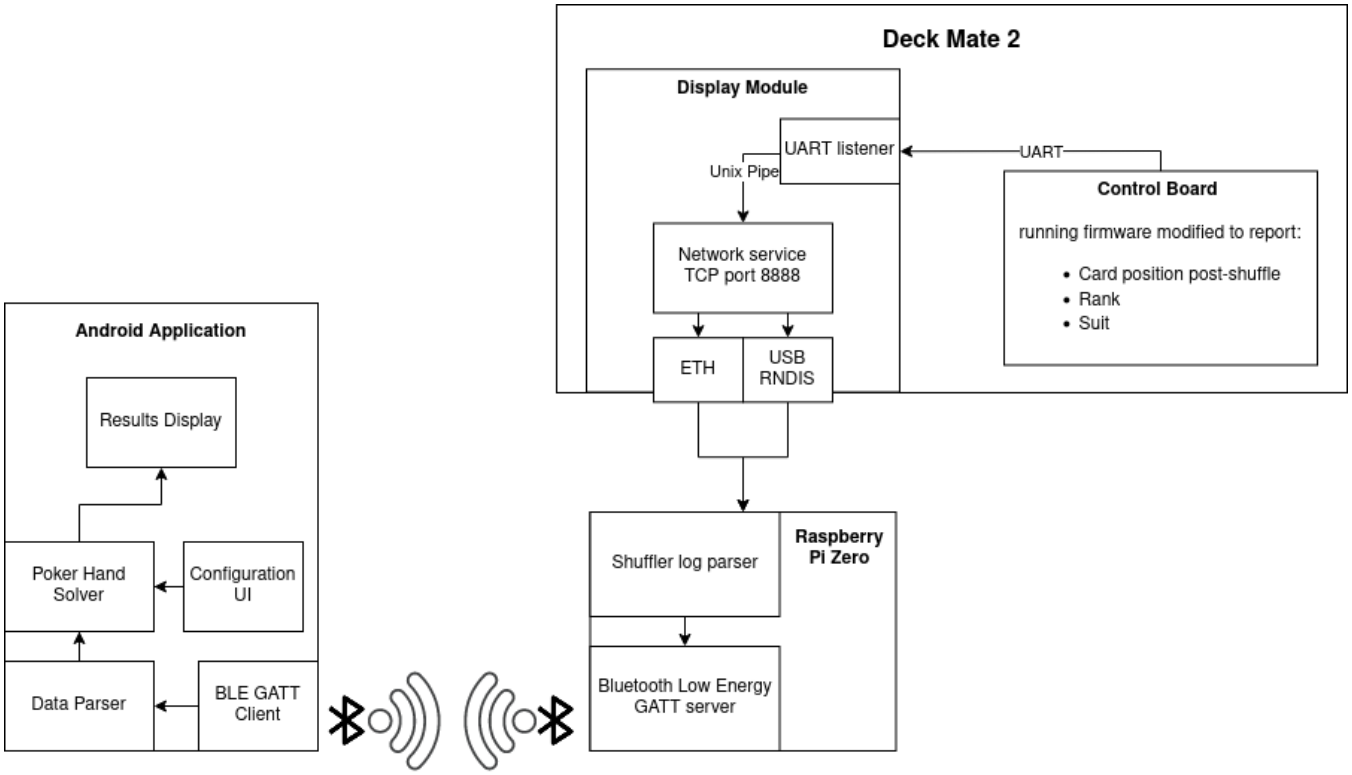


Figure 48. Architecture Diagram Including Cheating Apparatus

All the above techniques and implementations combine to allow for knowledge of full card order to be transmitted as soon as a shuffle is completed to the mobile app.

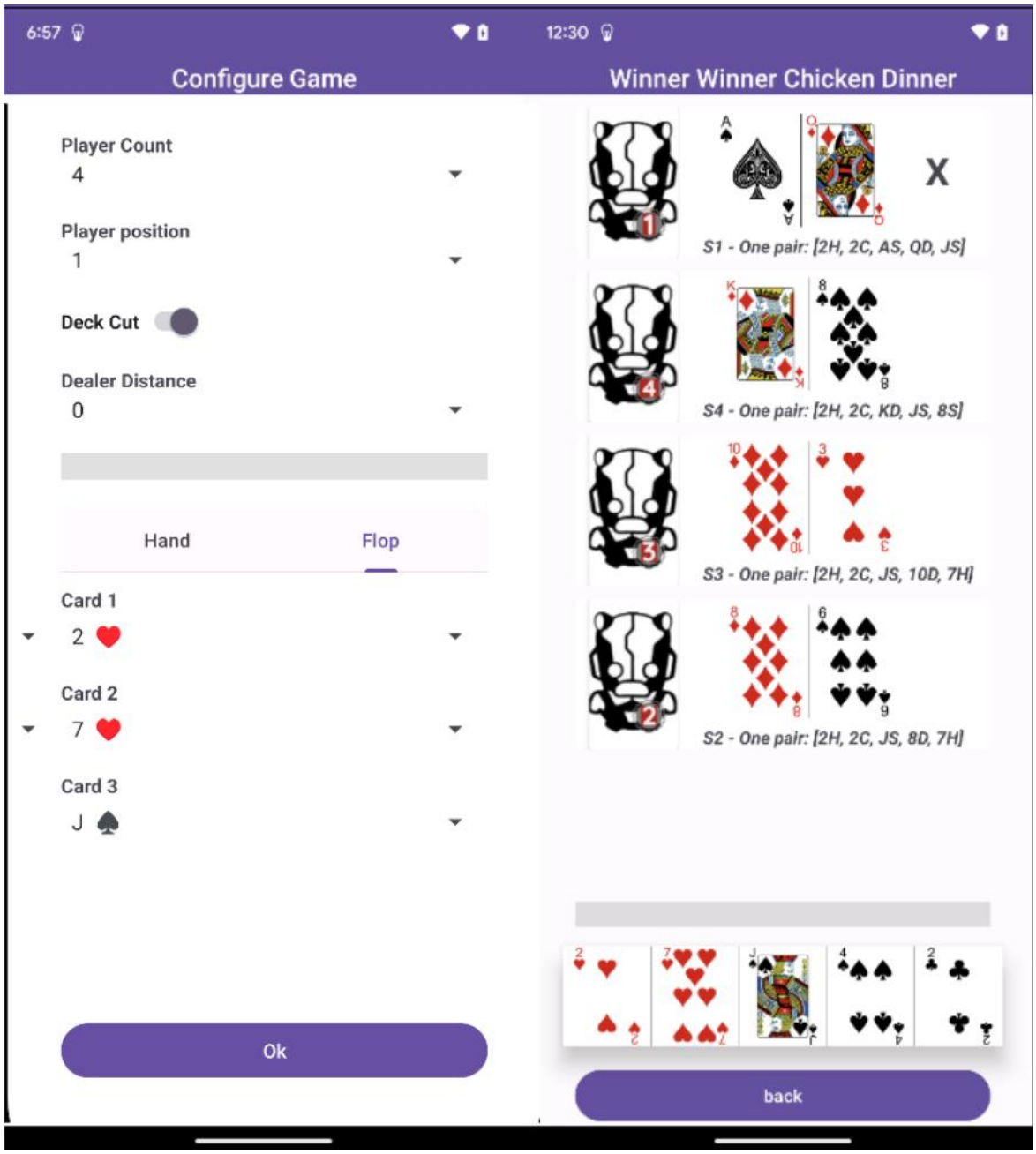


Figure 49. Mobile App Showing Configuration and Player Hands

The two screens in Figure 49 show the configuration for the hand, including entering of flop cards and indicating that the deck is being cut, and the hands of all players, in order from best to worst. In this case, we can see that the player at seat 1 will win with a one pair (twos, ace kicker), followed by seat 4 (twos, king kicker), then seat 3 (twos, ten kicker), and finally seat 2 (twos, eight kicker). The X marks the cheater's position for convenience as indicated in the configuration screen.

Execution

There are a few scenarios for how this mobile app could be used. In the first case, a third party cooperating with a cheating player at the table interacts with the app somewhere away from the table, either close by on

the rail or further away (within range of whatever communication medium is in use from the shuffler to the phone). The cooperator then uses traditional signaling techniques to communicate key information to the cheating player, such as whether or not they have the winning hand, whether certain bets are good or bad, etc.

Another option would be for the player to operate a version of the app that has been disguised to appear as something benign at the table. This scenario is less feasible, as the use of phones mid-hand is considered inappropriate and would be noticed by other players and possibly disallowed by the dealer. In games where the deck is not being cut (which is the case at low-stakes games in some card rooms, where cutting of the deck is sometimes forgone to increase hand rates and thus profits for the card room) no input is required from the cheater, and the cheater could simply glance at a phone screen briefly or even have the phone vibrate in their pocket to indicate a winning hand, making this scenario somewhat more feasible.

Deck Order Modification

In the same way the shuffler control board firmware was modified to log card placement information, modifications could be made to the firmware to introduce logic capable of placing cards in a predetermined or otherwise cheater-specified order. Though this technique was not implemented by IOActive during research for reasons that will be discussed below, the various scenarios above demonstrate that it would unquestionably be possible.

A few scenarios where control over deck order could be leveraged for cheating exist, with some caveats to each one. The largest issue with this kind of cheating is that if the cheater is inducing specific card orders repeatedly, statistical analysis of hands dealt for games in which cheating occurred would very quickly indicate that cheating was taking place. This is especially true if the cheater themselves is winning hands an unreasonable amount of the time.

The second major issue with directly controlling the order of the cards is the cut; because the cards will still be cut after they are shuffled, any attempts to force a win for a certain seat at the table, or even a certain set of hands, will be stymied. There are certain deck orders that have been worked out mathematically to always force a win for a certain seat, regardless of cut position,[5] but these are well-known and would be extraordinarily suspicious if they occurred even once, let alone several times.

Aside from simply attempting to force a win for a certain player, forcing a certain player to *lose* may also be a desirable outcome. This is less suspicious than the previous scenario to a point, but suffers from the same statistical deviation and would eventually be detected.

Finally, combining the two above scenarios, it may be possible to control card order to abuse what is known as a Bad Beat jackpot. This term refers to a special type of jackpot that is offered by some poker rooms or casinos, designed to reward players who experience an extremely unlikely and unfortunate outcome during a hand of poker. For example, a player with a highly favorable hand (four of a kind) loses to an even better, more unlikely hand (a straight flush). In this scenario, the casino will pay out what could be a rather large jackpot (generally tens of thousands of dollars on the high side) to all players at the table, favoring the losing bad-beat hand and the winning hand.

Controlling card order to construct this scenario with a group of accomplices all playing the same table is possible, where card order could be selected to attempt to estimate the cut position. Eventually, this cut

[5] https://www.benjoffe.com/holdem/

position guess would be correct, and the Bad Beat jackpot could be triggered; however, these jackpots are pooled over time and depleted upon payout, so this attack could only be conducted infrequently, requiring that jackpot pool be refilled before attempting it a second time. Furthermore, because this scenario is by definition extremely unlikely, conducting this attack multiple times at the same casino or using the same accomplices would likely be noticed as cheating by casinos and card rooms.

Ultimately, using the DM2 camera feature to exfiltrate the order of the deck to leverage for cheating is a much stealthier approach than using the features to create fixed decks.

Recommendations

Gaming operators can easily address concerns related to external bad actors by implementing physical restrictions on access to the exposed ports on the shuffler. This could be simply a metal enclosure wrapped around the shuffler where it is inserted in the table. This solution effectively mitigates potential threats from external attackers who have no affiliation with the operation; however, it is important to note that this measure does not fully address the broader issue of an internal bad actor who already has access to the devices.

One scenario could be for the manufacturer to add support for completely disabling features that are not being used. This would allow card rooms to not have to modify their tables and the shufflers themselves would have disconnected USB or Ethernet ports, for example.

For the players who are gaming in establishments that use these devices, it ultimately comes down to your trust in the gaming operator. Doug Polk, a professional poker player and operator of The Lodge card room shared some insight over these types of cheating scenarios in a Twitter thread.

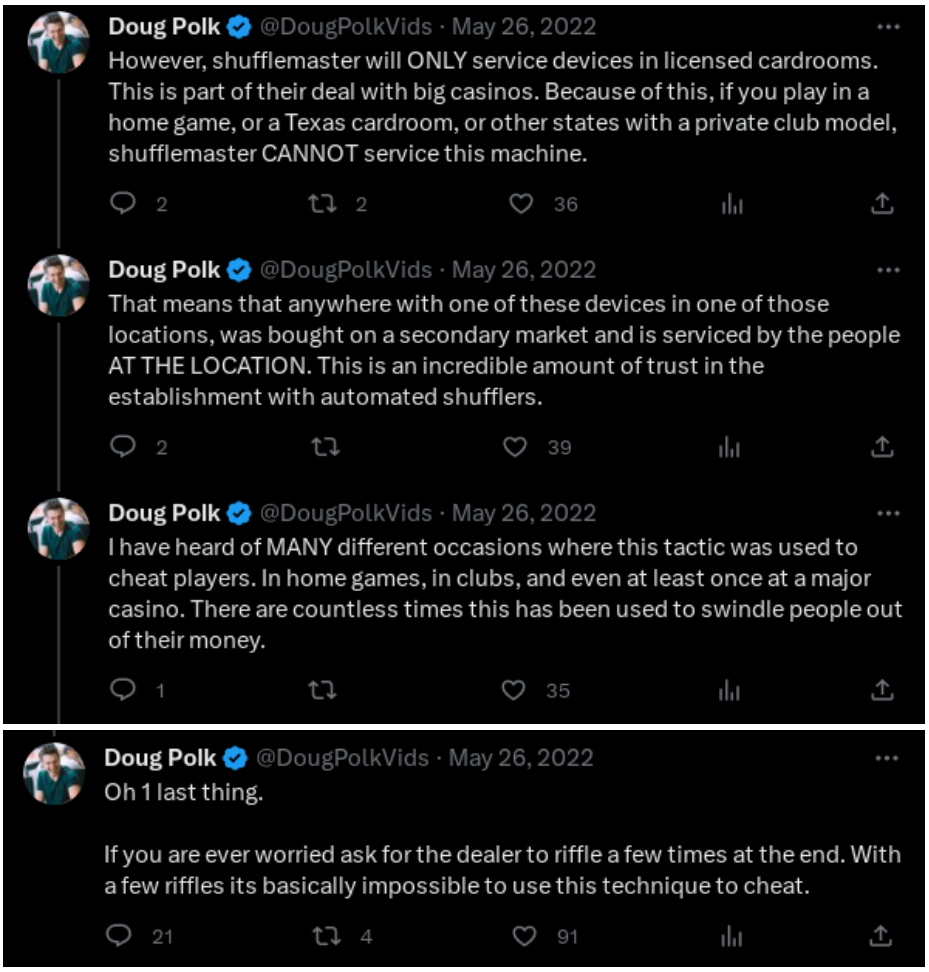


Figure 50. Doug Polk Twitter Thread[6]

[6] @DougPolkVids https://twitter.com/DougPolkVids/status/1529976301536280576

Conclusion

Overall, IOActive observed major insecurities and oversights in the design and architecture of the DM2. Based on the review of gaming standards and analysis of the scope of this research, we concluded that casino environments, like some Fortune 500 enterprise environments, can sometimes appear extremely hardened while being surprisingly fragile. IOActive demonstrated that once the firmware of the shuffler is compromised, the threat of cheating is real. This is true even for cases (such as with the DM1) where the device is incapable of determining the deck state.

Due to the unnecessarily exposed attack surface on the DM2 model, it is clear that the standard poker table design where the bottom of the shuffler exposed to players, needs to be changed. The threat model for the shuffler should expect an attacker to have access and implement defenses in this case, although the operators should also restrict access to the best of their abilities. This could be done by using a metal enclosure blocking access to the ports under the table or similar physical restriction. This improvement only provides mitigation against players but does not address the other attack scenarios described in this document, such as rogue maintenance employees.

More broadly, the security of both shuffler models was found to be well behind the state-of-the-art for secure devices. Basic security principals such as secure boot, adequate password policies, attestation, and various others addressed in this whitepaper were either entirely missing or badly implemented – these features represent the bare minimum bar for a vendor wishing to secure their device or system. The vendor of the shufflers examined in this whitepaper, as well as vendors of other gaming machines, must take steps to bring their products to modern security standards.

Furthermore, the Game Authentication Terminal protocol defined by the iGSA is not a secure method to authenticate software running on EGMs or peripherals. Auditors relying on this mechanism for integrity verification could be easily fooled by a compromised device. This G2S class must be reviewed by the iGSA and should be expanded with concepts from the Trusting Computing Group.

IOActive communicated the discovered vulnerabilities to the device manufacturer with the vulnerabilities being acknowledged and in a remediation process.

Finally, it is important to understand that a player's trust in the integrity of the poker game is dependent upon their trust in the operator. Concerned players should follow Doug Polk's recommendation and request the dealer to riffle the deck once removed from the shuffler, and as always, "protect your hand."

Appendix A: Additional Information

The purpose of this Appendix is to present the code utilized in this paper and provide relevant information about IOActive and the researchers' biographies.

DM1 Shuffling Algorithm in C

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define CARDS_IN_GAME 52

unsigned char CallCount;
unsigned short CardsRemaining;
unsigned short iterCards;
unsigned char TempDeck[CARDS_IN_GAME];
unsigned char RandomizedDeck[CARDS_IN_GAME];
unsigned char FinalDeck[CARDS_IN_GAME];
unsigned char Grips[CARDS_IN_GAME];

int bHasToInit = 1;
int bRandomizedDeckReady = 0;

unsigned short ProcessedCards = 0;
unsigned short CardsGripped   = 0;

#define SPECIAL_GRIP_VAL 0x02
unsigned short gComputedValueForUngrippedCards;


unsigned short GetRandom(int a, int b) {
    return rand() % b;
}

void GenerateRandomDeck() {

   if (bHasToInit == 1) {

        CallCount = 0;
        iterCards = 0;
        CardsRemaining = 0;
            bHasToInit = 0;

        for (int i = 0; i < CARDS_IN_GAME; i++) {
           TempDeck[i] = i;
           iterCards++;
        }
        //printf("TempDeck prepared!\n");

   } else {

      if (bRandomizedDeckReady) {
         printf("GetNextSeed()\n");
         return;
      }
```

```
      CardsRemaining = CARDS_IN_GAME - CallCount;
      //printf("CardsRemaining: %d\n", CardsRemaining);
      unsigned short r = GetRandom(0, CardsRemaining) & 0xFF;

      unsigned char pos = TempDeck[r];

      // Set the random position into the RandomizedDeck
      RandomizedDeck[CallCount] = pos;

      iterCards = r;
      r++;

      while (r < CardsRemaining) {
           pos = TempDeck[r];
           r++;
           TempDeck[iterCards] = pos;
           iterCards++;
       }

       CallCount++;

       if (CallCount > (CARDS_IN_GAME - 1))  {

        RandomizedDeck[CallCount] = TempDeck[0];
        bRandomizedDeckReady = 1;
       }

   }

}


void print_deck_array(char *array) {
    for (int i = 0; i < CARDS_IN_GAME; i++) {
        if (i > 0 && i%(13*2) == 0)
            printf("\n");
        printf("%02d ", array[i]);
    }
    printf("\n");
}


int gCardsGripped = 0;

void CalculateCardsToGrip() {

    int TmpCount = 0;

    for (int i = 0; i < ProcessedCards; i++) {

        if (RandomizedDeck[ProcessedCards] > FinalDeck[i]) {
            TmpCount++;
        }

    }
```

```
    gCardsGripped = TmpCount;

    printf("ProccessedCards:%d - Current pos: %d - gripped: %d\n",
            ProcessedCards, RandomizedDeck[ProcessedCards], gCardsGripped);

    if (gCardsGripped == ProcessedCards) {
        gComputedValueForUngrippedCards = 0;

    } else {

       int ungripped_cards = ProcessedCards - CardsGripped;

       gComputedValueForUngrippedCards = ungripped_cards * SPECIAL_GRIP_VAL;
    }
}

int main() {

    srand(time(NULL));

    GenerateRandomDeck();
    //print_deck_array(TempDeck);
    while(bRandomizedDeckReady == 0) {
       GenerateRandomDeck();
       //print_deck_array(TempDeck);
    }

    printf("Randomized Deck:\n");
    print_deck_array(RandomizedDeck);

    for (int i = 0; i < CARDS_IN_GAME; i++) {
        CalculateCardsToGrip();
        ProcessedCards++;
        FinalDeck[i] = RandomizedDeck[i];
        Grips[i] = gCardsGripped;
    }

    printf("Grips to perform:\n");
    print_deck_array(Grips);

    return 0;
}
```

DM2 Update Decryption Utility

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <openssl/sha.h>

struct ltv {
    uint16_t len;
    uint16_t type;
    char  data;
};

void hexdump(unsigned char *data, size_t size) {
    char ascii[17] = {0};
    size_t i;

    for (i = 0; i < size; ++i) {
        unsigned char c = data[i];
        size_t next = i+1;
        printf("%02X ", c);
        ascii[i % 16] = isprint(c) ? c : '.';
        if (next % 8 == 0 || next == size) {
            printf(" ");
            if (next % 16 == 0) {
                printf("|  %s \n", ascii);
            }
            else if (next == size) {
                size_t j;
                ascii[size % 16] = '\0';
                if (size % 16 <= 8) {
                    printf(" ");
                }
                for (j = size % 16; j < 16; ++j) {
                    printf("   ");
                }
                printf("|  %s \n", ascii);
            }
        }
        fflush(stdout);
    }
}

uint32_t walk(SHA_CTX *sha_ctx, char *key, char *iv, FILE *fptr, char **data, char
*digest) {
    uint32_t pos = 0;
    fseek(fptr, pos, SEEK_SET);

    struct ltv buf;

    uint32_t type1 = 0;
    uint32_t type2 = 0;
    uint32_t type3 = 0;
```

```
uint32_t type2_len = 0;

while (1) {
    fread((void *)&buf, sizeof(struct ltv), 1, fptr);
    pos += buf.len + sizeof(uint16_t);
    if (buf.type == 0x01) {
        type1++;
    } else if (buf.type == 0x02) {
        type2++;
        type2_len += buf.len-2;
    } else if (buf.type == 0x03) {
        type3++;
        break;
    }
    fseek(fptr, pos, SEEK_SET);
}
printf("Chunk counts:\n");
printf(" Type1: 0x%x\n Type2: 0x%x\n Type3: 0x%x\n", type1, type2, type3);
printf("\nTotal file len: 0x%x (%d)\n\n", type2_len, type2_len);

*data = (char *) malloc(type2_len);
char *enc_data = (char *) malloc(type2_len);
if (!*data) {
    perror("malloc()");
    return -1;
}
if (!enc_data) {
    perror("malloc()");
    return -1;
}
memset(*data, 0, type2_len);
pos = 0;
fseek(fptr, pos, SEEK_SET);
uint32_t offset = 0;
int len;

EVP_CIPHER_CTX *evp_ctx;
evp_ctx = EVP_CIPHER_CTX_new();

while (1) {
    fread((void *)&buf, sizeof(struct ltv), 1, fptr);
    buf.len -= 2; // adjust size since it includes header
    char * chunk_data = (char *)malloc(buf.len);
    if (!chunk_data) {
        perror("malloc()");
        return -1;
    }
    if (buf.type == 0x01) {
        // IV chunk
        fseek(fptr, pos + 4, SEEK_SET);
        fread((void *)chunk_data, buf.len, 1, fptr);
        printf("IV: ");
        hexdump(chunk_data, buf.len);
        memcpy(iv, chunk_data, 16);
        SHA1_Update(sha_ctx, iv, 16);
        EVP_DecryptInit_ex(evp_ctx, EVP_aes_128_cfb(), NULL, key, iv);
```

```
        }
        else if (buf.type == 0x02) {
            // data chunk
            fseek(fptr, pos + 4, SEEK_SET);
            fread((void *)chunk_data, buf.len, 1, fptr);
            SHA1_Update(sha_ctx, chunk_data, buf.len);
            EVP_DecryptUpdate(evp_ctx, *data+offset, &len, chunk_data, buf.len);
            EVP_DecryptFinal_ex(evp_ctx, *data+offset+len, &len);
            memcpy(enc_data+offset, chunk_data, buf.len);
            free(chunk_data);
            offset += buf.len;
        } else if (buf.type == 0x03) {
            // digest chunk
            fseek(fptr, pos + 4, SEEK_SET);
            fread((void *)chunk_data, buf.len, 1, fptr);
            memcpy(digest, chunk_data, buf.len);
            break;
        }
        pos += buf.len + 4;
        fseek(fptr, pos, SEEK_SET);
    }
    EVP_CIPHER_CTX_free(evp_ctx);
    return type2_len;
}

int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("Error: provide a path to an .enc1 file\nUsage: %s <path to .enc1
binary>\n", argv[0]);
        return 0;
    }
    FILE *fptr = fopen(argv[1], "rb");

    char *path = malloc(255);
    if (!path) {
        perror("malloc()");
        return -1;
    }

    memset(path, 0, 255);

    unsigned char key[16] =
{'R','E','D','A','C','T','E','D','R','E','D','A','C','T','E','D'};
    unsigned char iv[16] = {0};

    SHA_CTX sha_ctx;
    SHA1_Init(&sha_ctx);
    int i;

    unsigned char xor_1[16] = {0};
    for (i=0; i < 16; i++) {
         xor_1[i] = key[i] ^ 0x36;
    }
    SHA1_Update(&sha_ctx, xor_1, 16);

    char *data = NULL;
```

```
    char f_digest[SHA_DIGEST_LENGTH] = {0};
    uint32_t olen = walk(&sha_ctx, key, iv, fptr, &data, f_digest);
    if (data == NULL || olen == -1) {
        printf("walk error\n");
        printf("data: %p\n", data);
        printf("olen: 0x%x\n", olen);
        return -1;
    }
    unsigned char digest[SHA_DIGEST_LENGTH];
    SHA1_Final(digest, &sha_ctx);
    SHA_CTX f_ctx;
    SHA1_Init(&f_ctx);

    unsigned char xor_2[16] = {0};
    for (i=0; i < 16; i++) {
         xor_2[i] = key[i] ^ 0x5c;
    }
    SHA1_Update(&f_ctx, xor_2, 16);
    SHA1_Update(&f_ctx, digest, 4); // this is a minor bug due to its
implementation
    SHA1_Final(digest, &f_ctx);

    int ret = memcmp(digest, f_digest, SHA_DIGEST_LENGTH);
    if (ret) {
        printf("digest error! decryption likely failed\n");
    } else {
        printf("Calculated sha1 digest valid!: ");
        for (int i = 0; i < SHA_DIGEST_LENGTH; i++) {
            printf("%02x", digest[i]);
        }
        printf("\n");
    }
    snprintf(path, 255, "%s-data.bin", argv[1]);
    printf("writing to %s\n", path);
    FILE *optr = fopen(path, "wb");
    fwrite(data, olen, 1, optr);
    fclose(fptr);
    fclose(optr);
}
```

About IOActive

IOActive is a comprehensive, high-end information security services firm with a long and established pedigree in delivering elite security services to its customers. Our world-renowned consulting and research teams deliver a portfolio of specialist security services ranging from penetration testing and application code assessment through to semiconductor reverse engineering. Global 500 companies across every industry continue to trust IOActive with their most critical and sensitive security issues. Founded in 1998, IOActive is headquartered in Seattle, USA, with global operations through the Americas, EMEA and Asia Pac regions.

Visit www.ioactive.com for more information

Read the IOActive Labs Research Blog: http://blog.ioactive.com

Follow IOActive on Twitter: http://twitter.com/ioactive

About the Authors

Joseph Tartaro is a security researcher with over a decade of professional experience working on embedded systems. As a Principal Security Consultant at IOActive, he is primarily focused on vulnerability research, low-level code review, reverse engineering and development of specialized tooling. His past research covers a wide range of targets including video game consoles, causing havoc in the California DMV and exploiting vulnerabilities in platform security features running below the operating system. Joseph enjoys traveling the world to share his research at conferences such as Defcon, Black Hat, CCC, Ruxcon, hardwear.io and others.

Enrique Nissim is a security engineer with over 10 years of professional experience working on vulnerability research. As a Principal Security Consultant at IOActive, he is mainly involved in projects requiring a deep understanding of operating systems, CPU architectures, embedded firmware and software development. Over his career, Enrique has delivered multiple presentations at several leading events including CansecWest, Ekoparty, ZeroNights and Hardwear.io.

Ethan Shackelford is an Associate Principal Security Consultant at IOActive, with a specialization and particular interest in hardware and embedded security. He has researched a range of topics including various communication protocols, fault injection, and various cellular technologies. Outside of work, he enjoys finding new and interesting ways to repurpose hacked hardware, and developing binary reverse engineering techniques and tools.