Breath of the RF Field
Hacking Amiibo with Software-Defined Radio

James Chambers
@jamchamb_
INTRODUCTION
Amiibo
Amiibo

Lv. 28 Mario

Star Highway

Super Jump
Shocking Cape

Special

Select OK to send your FP back to the Amiibo and save your changes.

1:49.13

Feather Power

Jack

OK!
Amiibo

- Contain an NFC “tag”
- Near Field Communication (NFC)
  - Set of protocols for contactless power and data transfer between devices within a few centimeters of each other
- Amiibo use NTAG215
  - Part of the NTAG21x specification, a specific kind of NFC tag made by NXP Semiconductors
https://gototags.com/blog/whats-the-difference-nfc-tags-v-nfc-chips/
**Twilight Hack**

- Buffer overflow triggered with horse name, which is set by user and kept in save file
  - Save file edited using crypto keys obtained from hardware exploit
  - Edit save to smash the stack
- Used in software-based attack chain for installing homebrew software manager

See “Console Hacking 2008: Wii Fail” by Team Twiizers (fail0verflow)
Twilight Hack
Twilight Hack
Twilight Hack

```
OK.
Getting SD card status: 00000005
sd_reset_card(): got reply = 63000000
Card status: 00000700 [STANDBY]
Selecting card:
Card status: 00000900 [TRANSFER]
sd_set_blocklength(512)
sd_set_bus_width()
SD card detected
USBGecko serial interface detected
Loading FAT:
OEM Name: MSDOS5.0, 38 reserved sectors, FAT32.
Fat size = 1df200
FAT starts at sector 0x125
/ starts at sector 0x1f17
Reading boot.dol:
start cluster = 0000
start cluster = 0000
start cluster = 0003
start cluster = 0515
Found boot.dol!
done, filesize is 1588672
Box elf image at address 90100000
DOL image detected?
relocating 00000100 to 802191a0 (2016 bytes)
relocating 00000be0 to 80219980 (1586400 bytes)
clearing RSS (at 00000000, 0 bytes...)
```
Twilight Hack
**Twilight Hack**

- Like smartphones, modern game consoles put users in “jail”
- Meant to prevent piracy and cheating
  - Consoles often sold at a loss, in hopes of profiting from games and services
- No custom software allowed

Yifan Lu – “The 3DS Cryptosystem”
https://yifan.lu/2016/04/06/the-3ds-cryptosystem/
Splatoon 2 Series amiibo

Use the Splatoon 2 amiibo to unlock the outfits below! The Splatoon 1 variants of these amiibo unlock the outfits from Splatoon 1!

amiibo can also be used for storing loadouts and settings to use it on any Switch console when you use the amiibo! The original amiibo for Splatoon 1 can be used for this function as well!
Amiibo

• How easy would it to be to clone or spoof them?
• Can the data they store be used for “save game” style exploits?
• Could they be used to trigger an exploit on the Nintendo Switch?
GETTING STARTED
Getting Started

• What’s the data like?
NFC tag serial number

Settings, signatures, hashes

Character ID

Settings, signatures, hashes

Encrypted data
Getting Started

• How does the crypto system work?
Crypto
Details we do know:
• Amiibo data is encrypted
• Unique keys for each Amiibo
• Can’t copy data from one Amiibo to another
• AES and HMAC are involved
  • Encrypted data is also signed
Getting Started

Crypto

• amiitooll by socram8888 can perform encryption and decryption, but the crypto keys are kept private:
  • “Please note that keys used for signing and encryption are copyright of Nintendo and therefore they can't be shared, and I won't share post them here, send them using a private message, or anywhere else.”

• Often, the homebrew/game hacking community does not make crypto keys public to avoid enabling piracy.
Crypto

• Private online APIs for decrypting/encrypting Amiibo
  • Crypto keys stored on server, inaccessible to user

• Need to get an API key from the dev

• Activity on API is monitored

• Operations would be throttled by response time, possible rate limiting, and server downtime or decommission
  • Every payload attempt would require waiting for an API request
Crypto

- Amiibo cheat devices use a similar model
  - Online API performs all operations that require crypto
  - PowerSaves device has API authentication tied to hardware
    - “Reversing PowerSaves for Amiibo”
      http://blog.ghettoha.xxx/reversing-powersaves-for-amiibo/

- Cheat device manufacturers don’t want to reveal the crypto secrets, they want you to buy their device
How can we read, copy, and tamper with the data?
How can we simulate an Amiibo?
Proxmark 3
• SDR device for RFID and NFC
• Primarily used to research proximity ID cards
• Open source, reprogrammable microcontroller and FPGA
• Can be used to simulate a reader/writer or tag
GETTING STARTED

Fig 2. Block diagram of NTAG213/215/216
Proxmark 3
• Has basic ISO 14443A simulation
• Supports initial wakeup, identification, and selection (anti-collision)
• Can tell an NFC reader its serial number. That’s all.

Need to roll our own NTAG21x simulator.
To read, edit, and simulate Amiibo we need to:

- Make an NTAG215 simulator for Proxmark
- Obtain the Amiibo crypto keys
- Integrate Amiibo crypto operations with Proxmark simulator
CRYPTO CAVERNS
• Jailbreaks already existed for 3DS
  • New 3DS has built-in NFC/Amiibo compatibility
    • (watch the “Breaking the 3DS” CCC presentation!)
  • Reverse engineering the software will be much easier
    • Can get decrypted game and service binaries
    • Can tamper with processes
**Crypto Caverns**

- NTR CFW: Custom firmware for 3DS
- Adds a debugger to the 3DS
- Sadly, breakpoints don’t work
- Limited to peek & poke
  - Read and tamper with memory
  - View registers at random moments

---

**NTR CFW 3.3**  
- Process Manager  
- Enable Debugger  
- Set Hotkey  
* Take Screenshot  
  Real-Time Save (Experimental)  
  Power  
  Game Plugin

http://44670.org/ntr
• Try to dump and reverse engineer the NFC service to:
  • Find the crypto keys
  • Fully understand crypto system
  • Understand how games, NFC service, and Amiibo interact

• The encrypted or decrypted Amiibo images may be accessible in dumped memory

• Static analysis of binary
Crypto Caverns

- Wrote a custom homebrew app that uses the NFC service:
  - Scan for Amiibo
  - Read its content
  - Request app data, tag info (can’t request raw dump)
Plaintext 16-bit character strings: nickname, owner, Mii names
Game/Application data

Signature, settings, timestamps, write counter

Amiibo nickname; Registered owner Mii

Application ID, hash

Game/Application data

Signature, serial, character ID
• Loaded up the NFC service binary in Hopper
• Searched for intelligible strings
• Immediately found… something suspicious
  • “locked secret” and “unfixed infos”
• Need to figure out how these bytes are used
• Difficulties with static analysis
  • Poor automatic code identification due to the way the service binary is set up
  • No main entry point that calls down to all the other code
  • Series of service call stubs that call isolated functions
• Broken debugger, tons of unidentified code, disassembler crashes on Ctrl+F… what can I do?
Crypto Caverns

- Use existing reverse-engineered NFC service IPC calls from the homebrew library to figure out the locations of related code in the binary dump
• 0xC8A17620 “Invalid state” error code appears often when calling functions at the wrong time
**Crypto Caverns**

- Add new features to custom app to help identify code in the binary
- Version 1
  - Add a bunch of button bindings to trigger all known IPC calls
  - Connect debugger
  - Replace “invalid state” error code with 0xDEADBEEF and call random functions
  - Find where 0xDEADBEEF error shows up
• Version 2
  • Replace the “invalid state” error codes with address of the function that sets them
  • Button mash and find lots more addresses
Crypto Caverns

- Version 3
  - Programmatically rewrite select function prologues to immediately return address of function
  - Works even if not in invalid state
  - Observe function addresses in error codes
  - Navigate down found branch and select more functions to rewrite
`stub_nfcGetAmiiboSettings:`

```
00113800  push  {r4, lr}
00113802  mov    r4, r1
00113804  movs   r1, #0x11
00113806  lsls   r1, r1, #0x5
00113808  adds   r0, r0, r1
0011380a  ldrb   r0, [r0]
0011380c  cmp    r0, #0x2
0011380e  beq    stub_nfcGetAmiiboSettings+20

00113810  ldr    r0, = 0xc8a17600
00113812  pop    {r4, pc}
            ; endp

00113814  bl     usedBeforeAmiiboAPICall
00113818  mov    r1, r4
0011381a  bl     nfcGetAmiiboSettings
0011381e  pop    {r4, pc}
            ; endp
00113820  dd     0xc8a17600
```
Check last bit of ENVINFO in config memory

Pick developer or retail key info
HMAC Algorithm

Keys longer than blockSize are shortened by hashing them

```plaintext
if (length(key) > blockSize) then
    key += hash(key)  // Key becomes outputSize bytes long
```

Keys shorter than blockSize are padded to blockSize by padding with zeros on the right

```plaintext
if (length(key) < blockSize) then
    key += Pad(key, blockSize)  // pad key with zeros to make it blockSize bytes long
```

```plaintext
o_key_pad = key XOR [0x5c * blockSize]  // Outer padded key
i_key_pad = key XOR [0x36 * blockSize]  // Inner padded key
```

```plaintext
return hash(o_key_pad || hash(i_key_pad || message))  // Where || is concatenation
```
```c
int lots_of_amiibo_crypto(int arg0, int arg1) {
    sp = sp - 0x440;
    r5 = arg0;
    memcpy(sp + 0x2c, arg1 + 0x10, 0x24);
    memcpy(sp + 0x1b8, arg1 + 0x34, 0x20);
    memcpy(sp + 0x1e0, arg1 + 0x54, 0x2c);
    memcpy(sp + 0xc, arg1 + 0x80, 0x20);
    memcpy(sp + 0x50, arg1 + 0xa0, 0x168);
    var_1d8 = sp + 0x20c;
    memcpy(sp + 0x20c, arg1 + (0x41 << 0x3), 0x14);
    sub_11db8(sp + 0x220, 0x40);
    memcpy(sp + 0x240, arg1 + 0x60, 0x20);
    *(sp + 0x2c) = 0xa5;
    *(sp + 0x2d) = 0x0;
}
```

```c
void nfc3d_amiibo_tag_to_internal(const uint8_t *param) {
    memcpy(intl + 0x000, tag + 0x008, 0x008);
    // byte & lock bytes & capability container
    memcpy(intl + 0x008, tag + 0x000, 0x020);
    memcpy(intl + 0x028, tag + 0x010, 0x024);
    memcpy(intl + 0x04c, tag + 0x0a0, 0x168);
    memcpy(intl + 0x1b4, tag + 0x034, 0x020);
    memcpy(intl + 0x1d4, tag + 0x000, 0x008);
    memcpy(intl + 0x1dc, tag + 0x054, 0x02c);
    ...
```
• Tried to figure out how to use it with amiitool, but there isn’t enough data to build a full key file
• Using the content I did have, I tried searching for the keys online…
- Found some interesting notes…
- Found the whole key file…
- It contains a value not in the NFC service dump; what is it?
- What hasn’t shown up yet? The 3DS’s dedicated AES hardware

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVELOPER KEY INFO</td>
<td></td>
</tr>
<tr>
<td>&quot;unfixed infos&quot;, 0</td>
<td></td>
</tr>
<tr>
<td>&quot;locked secret&quot;, 0</td>
<td></td>
</tr>
<tr>
<td>RETAIL KEY INFO</td>
<td></td>
</tr>
<tr>
<td>&quot;unfixed infos&quot;, 0</td>
<td></td>
</tr>
<tr>
<td>&quot;locked secret&quot;, 0</td>
<td></td>
</tr>
</tbody>
</table>
Two key components are sent to AES engine
- “keyX” set by bootloader
- “keyY” set by firmware after it’s decrypted

Setting keyY triggers generation of the “normal” key

Normal key can be used to perform crypto operations in the engine, but can’t be read back out
“Key scrambling” algorithm performed by the AES engine was reversed by derrek, plutoo, smeamum; presented in “Breaking the 3DS” at CCC in 2015

All normal keys can be derived by applying this algorithm to the keyX and keyY components

AES crypto could then be performed without the 3DS hardware, but before that…
AES Engine
(CTR mode)

Key scrambler

Encryption key

Nonce || Counter

KeyX

KeyY

Nonce || Counter
AES Engine (CTR mode)

KeyX
KeyY

Key scrambler

Encryption key

Nonce || Counter

encrypt(key, nonce || counter + 0)
AES Engine
(CTR mode)

Key scrambler

Encryption key

Nonce || Counter

encrypt(key, nonce || counter + 0)  encrypt(key, nonce || counter + 1)
AES Engine
(CTR mode)

Encryption key

Nonce || Counter

encrypt(key, nonce || counter + 0)
encrypted(key, nonce || counter + 1)
encrypt(key, nonce || counter + 2)
AES Engine (CTR mode)

Key scrambler

Encryption key

Nonce || Counter

encrypt(key, nonce || counter + 0)
encrypt(key, nonce || counter + 1)
encrypt(key, nonce || counter + 2)

plaintext buffer
AES Engine (CTR mode)

Nonce || Counter

Encryption key

Nonce || Counter

Encrypt(key, nonce || counter + 0)
Encrypt(key, nonce || counter + 1)
Encrypt(key, nonce || counter + 2)

plaintext buffer
AES Engine (CTR mode)

Key scrambler

Encryption key

Nonce || Counter

plaintext buffer

⊕

ciphertext

encrypt(key, nonce ||counter + 0)

encrypt(key, nonce ||counter + 1)

encrypt(key, nonce ||counter + 2)
A ⊕ 0 = A
AES Engine (CTR mode)

Key scramble

Encryption key

Nonce || Counter

KeyX

KeyY

encrypt(key, nonce || counter + 0)

encrypt(key, nonce || counter + 1)

encrypt(key, nonce || counter + 2)

⊕

zeros
KeyX
KeyY
Nonce || Counter

AES Engine
(CTR mode)

Key scrambler

Encryption key

Nonce || Counter

encrypt(key, nonce || counter + 0) ⊕ encrypt(key, nonce || counter + 1) ⊕ encrypt(key, nonce || counter + 2) = zeros

encrypt(key, nonce || counter + 0) encrypt(key, nonce || counter + 1) encrypt(key, nonce || counter + 2)
• A XOR pad can be used to perform CTR mode encryption and decryption without access to the AES engine hardware
• Only way to do crypto externally until hardware AES engine’s key scrambling algorithm was cracked
• Wrote a Python script for trying out a list of different keys from the hardware keys dump, with the ability to adjust AES-CTR internals
• Try encrypting zeros with different keys, nonce, and counters based on NFC service dump values until the known good XOR pad result is found
• With the following setup, it works:
  • The encryption key is the composite NFC service key from AES chip
  • The nonce and initial counter values are from the NFC binary
  • The counter is in big endian representation during AES-CTR
• Mysterious value is 32 byte XOR pad using NFC-specific key
The Amiibo Crypto system
Amiibo data has two partitions:
1. Tag (“locked secret”)
   • NTAG215 serial number
   • Character ID
   • Unique 32 byte sequence
2. Data (“unfixed infos”)
   • Settings, bit flags
   • Registered owner Mii data
   • Game/application data
The Amiibo Crypto system
1. Generate two sets of AES-CTR parameters and HMAC keys with a Deterministic Random Bit Generator
   • One set per partition
2. Sign each partition with its generated HMAC key
3. Encrypt data partition with AES-CTR using its generated AES key, nonce, and counter
   • Tag is stored plaintext; ignore that set of AES values
4. Rearrange the buffer for storage in NFC tag EEPROM

Reverse the process to decrypt (rearrange buffer, generate keys, decrypt data partition, check HMACs)
Amiibo
- Write counter
- Serial number
- Unique 32 bytes

NFC Service
- AES parameters
- HMAC parameters
- Partition name
- Partition magic bytes

3DS AES engine
Encrypt unique 32 bytes

HMAC Deterministic Random Bit Generator
- type string || (write counter) magic bytes || serial || Encrypted unique 32 bytes

Amiibo unique signing and encryption keys
- AES key || AES nonce || AES counter || HMAC key
You got the **Amiibo crypto system**! It’s incredibly convoluted!
Proxmark Plateau
Proxmark Plateau

- Use the Proxmark sniffer and data sheets to figure out protocol
- NXP’s NTAG21x data sheet is public
- Proxmark traffic display already recognizes many common ISO 14443 and NFC tag commands
  - Not always compatible with specific tag in use; may need changes
Implemented in ISO14443a simulator

Specific to NTAG21x
Proxmark Plateau

• Commands to implement:
  • READ
  • FAST_READ
  • WRITE
  • GET_VERSION
  • PWD_AUTH
  • READ_SIG

• Buffer for tag EEPROM contents
State of the codebase: spaghetti
A lot of the work is just figuring out how everything is set up
  There’s no developer documentation
Modular, self-contained NTAG21x logic
Weld NTAG215 simulator on to ISO 14443a simulator
Questions and Answers

The syntax of an if...else statement in a programming language:

```
if(boolean_expression):
    ok
else:
    ok
```

Example:
```
ok
```

EDIT

```
Rdr | 52 |
---+----|
| Tag | 44 00 |
| Rdr | 93 20 |
| Tag | 88 64 3d 36 87 04 3d 36 87 11 10 |
| Tag | 84 da 17 |
| Rdr | 95 20 |
| Tag | 4a b3 49 81 31 |
| Rdr | 95 70 4a b3 49 81 31 fc 67 |
| Tag | 88 fe 51 |
| Rdr | 68 f8 32 |
| Tag | 90 04 94 02 01 00 11 03 01 9e |
| Tag | 90 30 99 b9 |
| Tag | f1 10 ff ee a5 08 03 00 4a 0b 34 18 05 4e 3a 06 1f |
| Tag | b2 d9 d0 a9 67 5c 60 |
| Tag | 80 80 9a 16 |
| Rdr | 3a 00 3b 90 df |
| Tag | 94 36 37 4a b3 49 81 31 48 0f e9 f1 10 ff ee |
| Tag | a5 00 06 b3 0b b3 18 05 4e 3a 06 a1 b7 0b 96 |
| Tag | 9f 4a d4 71 0c 0d 62 df ba 56 e1 ea 2c 4c cb |
| Tag | c5 8f 7e 2b 5d 4a 62 8e a2 02 8d 97 1f 5b 4b e4 |
| Tag | 27 b9 76 16 c7 12 f1 7b dc 31 18 e5 be 37 75 48 |
| Tag | 53 56 68 07 19 19 00 00 00 00 02 8d 12 b5 04 |
| Tag | 1a 90 97 f4 d0 17 87 08 42 ff e5 01 54 2d b1 63 |
| Tag | f1 bf e9 7a 96 f6 6a 6a 3c 82 69 58 c5 0d 45 73 |
| Tag | 85 20 9e 27 9e 54 69 78 cl ca b4 25 50 9d 8b b4 |
| Tag | 67 f6 24 23 8d ed fd 84 af cc 6d 74 cd 07 db 2e |
| Tag | 70 a6 32 53 26 df 03 6e 14 30 fe a4 3f 42 fa de |
| Tag | 15 f6 34 10 55 60 46 a1 3a 7b 29 71 e4 a6 9a 67 |
| Tag | 35 s5 c8 e4 4c 9b 47 b0 f8 25 f8 a2 b8 c6 90 df |
| Tag | 83 6d 51 5a aa 1e 2b 53 b8 35 65 fe a1 f0 c9 1e |
| Tag | 85 b2 59 58 03 38 42 21 d2 63 cc dd dc cb 84 e6 |
| Tag | e9 d3 |
GET_VERSION

- “The GET_VERSION command has no arguments and replies the version information for the specific NTAG21x type.”
- NTAG215 info: 00 04 04 02 01 00 11 03 (+CRC-16)
- Add pre-cached response with 16 bit CRC value
  - Cyclic Redundancy Check: used for detecting transmission errors
  - Pre-caching necessary for low latency commands
  - Modulation must be calculated for data bytes before the timeout (usually 5ms)
• Read four pages, beginning at specified page
  • A page is a four byte block of data. 16 bytes are returned.
  • An NTAG215 has 135 pages (540 bytes total)
• Only used to read the capability container in page 3, so we don’t need to implement everything (locked pages, rolling over after end of memory, etc.)
• Biggest hurdle is implementing a memory buffer to store the EEPROM data
• Get buffer dynamically, calculate CRC-16, and send it back
Tag EEPROM Buffer

• The Proxmark has a “big buffer” with custom malloc
• Undocumented card memory section that can be preserved while freeing the rest of the memory
• Can be populated from client using USB commands
• Need to set up interface in Lua to handle reading/writing to card memory buffer
• Update simulator to use card memory buffer when simulating NTAG215
FAST_READ

- Read from given start page to end page
- Reading all pages at once is allowed, but the 3DS only reads up to 60 pages at a time
  - Uses three separate FAST_READ commands to get all memory
- Should send a NAK if request is out of range, but the consoles never do this
  - Still implement some bounds checks to prevent crashes
  - Poor signal results in noisy, corrupted commands
- Get buffer, calculate CRC-16, send it back
Proxmark Plateau

WRITE

• Write to a single page
• Update 4 bytes in the card memory buffer and send ACK
• Could potentially maintain state to see whether password unlock was used, but it’s not necessary at the moment
PWD_AUTH

• 32 bit password for unlocking write capability
  • One of NTAG21x’s built-in security features
• After 7 failed attempts, pages will be permanently locked
• Respond with 80 80 (+CRC-16) to accept password
• We can simply accept any password for the simulator
  • But it would still be interesting to know how passwords are calculated for more accurate simulation, or for writing to a real Amiibo
**Proxmark Plateau**

**PWD_AUTH**

- With SDR you can supply an arbitrary serial number and get back the password when the console attempts to authenticate
- Serials 00000000000000 and 04000000000000 give back AA55AA55...

<table>
<thead>
<tr>
<th>Serial / UID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 00 00 00 00 00 00</td>
<td>AA 55 AA 55</td>
</tr>
<tr>
<td>04 AA 00 00 00 00 00</td>
<td>00 55 AA 55</td>
</tr>
<tr>
<td>04 AA 55 00 00 00 00</td>
<td>00 00 AA 55</td>
</tr>
<tr>
<td>04 AA 55 AA 00 00 00</td>
<td>AA 00 00 55</td>
</tr>
</tbody>
</table>

- It’s based on XORing serial number bytes
  - $A \oplus 0 = A$; $A \oplus A = 0$
### Proxmark Plateau

**PWD_AUTH**

<table>
<thead>
<tr>
<th>Serial/UID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 FF 00 00 00 00</td>
<td>55 55 AA 55</td>
</tr>
<tr>
<td>04 00 FF 00 00 00</td>
<td>AA AA AA 55</td>
</tr>
<tr>
<td>04 00 00 FF 00 00</td>
<td>55 55 55 55</td>
</tr>
<tr>
<td>04 00 00 00 FF 00</td>
<td>AA AA AA AA</td>
</tr>
<tr>
<td>04 00 00 00 00 FF</td>
<td>AA 55 55 55</td>
</tr>
<tr>
<td>04 00 00 00 00 00 FF</td>
<td>AA 55 AA AA</td>
</tr>
</tbody>
</table>
# Proxmark Plateau

**PWD_AUTH**

<table>
<thead>
<tr>
<th>Serial/UID</th>
<th>Password (XOR AA55AA55)</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 FF 00 00 00 00 00</td>
<td>FF 00 00 00</td>
</tr>
<tr>
<td>04 00 FF 00 00 00 00</td>
<td>00 FF 00 00</td>
</tr>
<tr>
<td>04 00 00 FF 00 00 00</td>
<td>FF 00 FF 00</td>
</tr>
<tr>
<td>04 00 00 00 FF 00 00</td>
<td>00 FF 00 FF</td>
</tr>
<tr>
<td>04 00 00 00 00 FF 00</td>
<td>00 00 FF 00</td>
</tr>
<tr>
<td>04 00 00 00 00 00 FF</td>
<td>00 00 00 FF</td>
</tr>
</tbody>
</table>
xormask = '\xaa\x55\xaa\x55'
for i = 0 to 3:
pwd[i] = xor(uid[i+1], uid[i+3], xormask[i])
**Amiibo integration**

- Proxmark client has a Lua script interface
  - Can write and update Lua programs without recompiling client
  - Easier to implement, has OOP-ish features
- Wrote Lua wrapper for amiitool and compiled as a library
- Amiibo object in Lua
  - Uses amiitool Lua module for packing and unpacking data
- Card memory interface for populating and reading card memory
- Features
  - Dump and unpack data from Amiibo
  - Load and simulate image
  - Save/restore state of card memory
Proxmark Plateau

Proxmark3
- NTAG215 Simulator
  - EEPROM buffer

Proxmark client
- amiibo.lua
  - Fuzzing logic
  - amiitool Lua wrapper

USB
You got the NTAG215 simulator!
Why does it smell like spaghetti?
• Works fine on the 3DS, but when the Switch finally comes out in March:
  • ECDSA signature of the tag’s serial number is actually checked with the READ_SIG command
    • The Wii U also uses this check, but I hadn’t been testing on it
  • The ID card shaped Proxmark HF antenna works very poorly with the Switch

• It’s not very effective.
ANTENNA PROBLEMS
Antenna Problems
Antenna Problems
Antenna Problems

Official Proxmark3 instructions: rip open cable and wrap it around until it works
Making an Antenna
Making an Antenna
Making an Antenna

• Fail: Signal is not strong enough to read tags or be picked up by console.
  • Got 0.1 V, need around 12V.

• Can’t just buy any “13.56 MHz antenna” and attach it: tuning must factor in the rest of the device
  • Antenna forms inductor-capacitor circuit with capacitor on the Proxmark3 board
  • Capacitor has a capacitance of 47 picofarads (pF). Need to adjust this to match the inductance of a different antenna.
## Making an Antenna

### 2. Specifications

<table>
<thead>
<tr>
<th>Flexible PCB Near-Field Communications Antenna</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Frequency</strong></td>
</tr>
<tr>
<td><strong>Inductance @ 13.56 MHz</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mechanical</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Antenna Dimensions</strong></td>
</tr>
<tr>
<td><strong>RoHS Compliant</strong></td>
</tr>
<tr>
<td><strong>Adhesive</strong></td>
</tr>
<tr>
<td><strong>Weight</strong></td>
</tr>
</tbody>
</table>
Making an Antenna

- Current resonant frequency: \( \frac{1}{2\pi \sqrt{15.9 \, \mu H \times 47 \, pF}} = 5.822 \, MHz \)
- Desired resonant frequency: 13.56 MHz
- Solving for capacitance with desired frequency: need 8.67 pF.
- Need to reduce capacitance without changing Proxmark board.
Making an Antenna

• Reduce capacitance by adding capacitors in series:

\[
\frac{1}{\frac{1}{47\text{pF}} + \frac{1}{C_2} + \frac{1}{C_3} + \cdots + \frac{1}{C_n}} = 8.67\text{pF}
\]

• Ideal: add a 10.6 pF capacitor. Hard to order one that specific though.
• Alternative: get closest approximation with a series of whole-number capacitance levels available in small quantities online
Making an Antenna

```
[ jchambers@LT8220 ~/Documents/amiibo ]
$ ./caps.py 8.67 --included 47 --allowed 9 10 11 12 15 16 18 20 22 24 27 30 33 36 39
Closest set of 1 caps gives 8.913793pF
[47.0, 11.0]
Closest set of 2 caps gives 8.641471pF
[47.0, 36.0, 15.0]
Closest set of 3 caps gives 8.679351pF
[47.0, 33.0, 33.0, 30.0]
Closest set of 4 caps gives 8.074890pF
[47.0, 39.0, 39.0, 39.0, 39.0]
Closest set of 5 caps gives 6.689781pF
[47.0, 39.0, 39.0, 39.0, 39.0, 39.0]
Closest set of up to 5 caps gives 8.679351pF
[47.0, 33.0, 33.0, 30.0]
[ jchambers@LT8220 ~/Documents/amiibo ]
$ 
```
Making an Antenna

- 13.9 MHz resonant frequency with spare 10pF capacitor
- Boosts strength from 0.1 to 7V @ 13.56 MHz
Making an Antenna

proxmark3> script run amiibo
--- Executing: amiibo.lua, args ''
Loaded retail keys from all_in_one_keys.bin
Tag type: NXP MIFARE Plus 2K
Tag UID: 043d36874ab349813148
Tag len: 540
Figure ID: 19190000000900020d
Settings init: true
Nickname: Peekandpwn
Appdata writes: 60
UID: 043d364ab34981
Write key: ddd0a967

-----Finished
proxmark3> hw tune

Measuring antenna characteristics, please wait....#db# DownloadFPGA(len: 42096)

# LF antenna: 0.00 V @ 125.00 kHz
# LF antenna: 0.00 V @ 134.00 kHz
# LF optimal: 0.00 V @ 12800.00 kHz
# HF antenna: 6.91 V @ 13.56 MHZ
# Your LF antenna is unusable.
# Your HF antenna is marginal.
proxmark3>
1.3 Security

- Manufacturer programmed 7-byte UID for each device
- Pre-programmed Capability container with one time programmable bits
- Field programmable read-only locking function
- **ECC based originality signature**
- 32-bit password protection to prevent unauthorized memory operations
ECDSA signature of tag’s serial number
  • Elliptic Curve Digital Signature Algorithm

Signature proves tag originates from NXP Semiconductors

Prevents simple counterfeiting
  • Can’t make signature for new serial without NXP private key

Ideal benefit: A valid Amiibo NFC tag must be produced by NXP and initialized by Nintendo. Amiibo data is bound and unique to the tag it’s created for.
  • No one else can produce or duplicate Amiibo.
• Can’t use an arbitrary serial number without the ECDSA signature created with NXP’s private key

• Implications for simulating, fuzzing:
  • Some games limit usage by serial number and time (Breath of the Wild)
  • Need to amass a collection of serial/signature pairs…
  • or rewrite with the same serial number and redo encryption each time
  • Can’t simulate with arbitrary serial number 😞

• … or can you?
How do cheat devices do it?

• PowerSaves offers “serial randomization” cheat for bypassing rate limit in Breath of the Wild
  • Can only scan an individual Amiibo once per in-game day

• Cheat devices could harvest serial/signature pairs from users’ Amiibo…
  • Haven’t verified this, it’s just a possibility
  • Amiibo image uploaded to API for each operation
Crypto implementation errors?

• Reusing the same “random number” in ECDSA will compromise the private key
  • See fail0verflow’s PlayStation 3 hacking talk
• Used example SDK code to parse out the nonce parameter from signature
• They don’t appear to reuse the nonce
  • in the small sample I checked
Protocol or application logic flaws?

Verification process:
1. Select tag by ID (anti-collision)
2. Read NXP tag signature
3. Validate selected ID with signature
4. Read entire image from EEPROM
5. Generate keys for image
6. Perform HMAC validation of image

Can you spot the bug?
Protocol or application logic flaws?
Verification process:
1. Select tag by ID (anti-collision)
2. Read NXP tag signature
3. Validate selected ID with signature
4. Read entire image from EEPROM
5. Generate keys for image
6. Perform HMAC validation of image
**Originality Check**

- Supply any known ID and signature pair during selection
- Any Amiibo data can be returned through the read commands afterwards, regardless of the contained tag ID
  - Patched on Switch in 5.0.0; no fix for Wii U
- **No crypto keys necessary**
  - Don’t need to do any crypto rewriting to load an image
  - Arbitrary serial number in tag data
Future Work

- Finish antenna for Switch
- Examine new layer of encryption added to application data in games like Splatoon 2
- Create test harness for fuzzing on the Switch, 3DS
Tool Release

Get amiimikyu at
https://github.com/nccgroup/proxmark3-amiimicyou

amiitool Lua wrapper at
https://github.com/jamchamb/amiitool
Thanks to...

• Jeff Dileo
• Nolan Ray