@\_Tim\_: could you try writing an amiibo with a couple pages different from normal? For page 0x02, make the last two bytes 0 (to disable static locking), leave the capability container (page 0x03) default (0xE1 0x10 0x3E 0x00) (the values on 3DSbrew don't really make sense given the spec), leave page 0x82 0 (to disable dynamic locking). Set page 0x83 to 0x00 0x00 0x00 0xFF to disable password checking, and page 0x84 to 0x10 0x00 0x00 0x00 to disable the config lock, count protection, and auth limit. I'm curious if the Wii U will read an Amiibo with all of the protection disabled (and a more standard capability container), or not (I suspect it checks at least some of these). I find it very interesting that the last byte of the capability container is 0xEE as 0xE is a proprietary value according to the spec (and the first byte being 0xF1 makes no sense as the spec says 0xE1 is a magic value).

“\_tim\_：你能试着写几页不同于正常的amiibo？页面0x02，让最后的两个字节0（禁用静态锁定），离开能力容器（页0x03）默认（0xe1 0x10 0x3e 0x00）（在3dsbrew也值没有真正意义的规格），离开页面0x82 0（禁用动态锁定）。设置页面0x83为0x00 0x00 0xff禁用密码检查，并0x84 0x10 0x00 0x00 0x00到网页禁用配置锁，计数保护和授权限制。我很好奇，如果Wii U将读amiibo所有的保护残疾人（和更标准的能力，集装箱）或不（我怀疑它检查至少其中一部分）。我觉得很有趣的能力容器的最后一个字节是0xee作为0xe根据规格专有的价值（和第一个字节被0xf1毫无意义的规范说0xe1是魔法值）。

One other small problem, I'm not sure how to generate the 32 byte xorpad. The pastebin post a while back with all the keys in it had the AES IV and KeyY needed to create the xorpad, and I feel like it should be fairly straight forward with Decrypt9, but I'm not really that familiar with xorpad generation. Has anybody else done this before?

另外一个小问题，我不知道如何生成32字节的xorpad。引擎收录后，所有的钥匙都AES需要创建xorpad IV和刻意，我觉得它应该是相当直接的与decrypt9，但我并不熟悉xorpad代。以前有人做过这个吗？

Set1 keys:

1st 3 are DEVs data (and they are all correct).

2nd 3 are DEVs data and HMAC key is 830E75...

Set2 keys:

1st 3 are retail data and HMAC key is 1D164B...

2nd 3 are retail data and HMAC key is 7F752D...

As stated, algo for xorpad: AES-128 CTR mode: 1st key is AES KeyY, 2nd key is AES IV.

Resulting xorpads are:

495B197A5B802055AEE8AE8EA08E053233C770A8A99E6DAD6F1CA01FE3618022 (for DEVs)

044917DC76B49640D6F83939960FAED4EF392FAAB21428AA21FB54E545054766 (for retail)

1B+4bytes-PWD+2bytes ISO14443A-CRC (7 bytes total).

and you should get 2bytes-PACK back as answer if the command got executed correctly.

I suggest you to find and app that is able to manage ALL the NTAG215 command set (not only ISO14443A standard commands because 1B command is not standard, it is NXP proprietary) or to send the raw command with or without automatically calculating the ISO14443A-CRC.

xorpads ARE NOT copyrighted materials... they are just... xorpads, so no reason to censor them;)

Everything started fromthisgreat job.

你应该得到2bytes包回答如果命令被正确执行。

我建议你找到应用程序，可以管理所有的ntag215命令集（不仅因为1b ISO14443A标准命令的命令是不标准的，它是NXP专有的）或送原命令或不自动计算iso14443a-crc。

xorpads不受版权保护的材料…他们只是…xorpads，所以没有理由的审查；）

一切都是从这份伟大的工作开始的。

Alright, so I'm equipped with everything I need to give this a shot today:

- blank NTAG215 tags

- Amiibo dump

- hex editor

- the key to encrypt it

- Android smartphone with AmiiWrite

1) I decrypted the Amiibo dump

2) I read the NTAG215 with Android app NFC TagInfo to get the 7-byte UID.

3) I opened the Amiibo dump in a hex editor...

Where is the location of the UID to change it?

fiveighteen said:↑

Alright, so I'm equipped with everything I need to give this a shot today:

- blank NTAG215 tags

- Amiibo dump

- hex editor

- the key to encrypt it

- Android smartphone with AmiiWrite

1) I decrypted the Amiibo dump

2) I read the NTAG215 with Android app NFC TagInfo to get the 7-byte UID.

3) I opened the Amiibo dump in a hex editor...

Where is the location of the UID to change it?

Click to expand...

First decrypt Amiibo and go to offset 0x1D4 (8 bytes UID BCC0)

So the NTAG215 UID is 7 bytes, but the Amiibo UID is 8 bytes? How does that work - pad the end with 00?

The UID is the first seven bytes of the encrypted dump, so you can search it the decrypted dump and change it.

EDIT: javiMaD is right. The location of the UID in the decrypted dump starts at 0x1d4. Also, are you using the API or amiitool?

NTAG 215 UID:

04 C3 7A 52 C2 3E 80

DUCK HUNT DUO UID:

04 FC 30 40 82 03 49 80

It concerns me that they both start with 04 and end with 80 but there's an extra byte in the middle of the Amiibo.

Should I be changing Duck Hunt Duo to:

04 C3 7A 52 C2 3E 80 80 ??

Extra byte is BCC0 = 0x88 xor UID0 xor UID1 xor UID2

88 xor 04 xor C3 xor 7A = 35

04 C3 7A3552 C2 3E 80

Oh, that's what you meant. I apologize for not understanding. Thanks so much for the info

The first 10 bytes of the tag are determined by the NFC type A spec.

0: UID0 - the manufacturer code, always 0x04 for any NXP tag

1: UID1

2: UID2

3: BCC0 - CT ^ UID0 ^ UID1 ^ UID2 (CT is 0x88)

4: UID3 - Can't be CT (0x88)

5: UID4

6: UID5

7: UID6

8: BCC1 - UID3 ^ UID4 ^ UID5 ^ UID6

9: Internal - Always 0x48

Note that you can't actually write to any of these bytes on an actual NTAG, the first two pages are locked at the factory, and trying to write values to the first two bytes of the 3rd page does nothing (thus the static lock bytes can be set without worrying about the first two bytes of the write). Also note, that tags with 4 or 10 byte UIDs have a different layout (all NTAG21x tags have 7 byte UIDs).

注意，你不能在一个实际NTAG写任何这些字节，前两页锁在工厂，并尝试写值到第三页的第一个字节（因此没有静态锁定字节可以不用担心写的第一个字节）。还要注意，标签有4或10字节的流体有不同的布局（所有ntag21x标签有7字节的UID）。

PokeAcer said:↑

EDIT: I'm essentially asking if someone knows an easy way to calculate that extra byte.

(example for the UID I was using)

Extra byte UID3:

= 0x88 xor UID0 xor UID1 xor UID2

= 88 xor 04 xor C3 xor 7A

= ((10001000 xor 00000100) xor 11000011) xor 01111010

= (10001100 xor 11000011) xor 01111010

= 01001111 xor 01111010

= 00110101

= 35

XOR Truth Table:

AB | Q

00 | 0

01 | 1

10 | 1

11 | 0

PokeAcer said:↑

My head hurts from this xD

If someone can make a way to calculate this either online or as a program, I swear I will <3 you forever

You can usehttp://xor.pw/and the same logic as my above post.

88 xor UID0 = VAL1

VAL1 xor UID1 = VAL2

VAL 2 xor UID2 =UID3

Ex:

88 xor 04 = 8C

8C xor C3 = 4F

4F xor 7A =35

PHP (which is the worst langauge ever) actually supports xor, it is literally just "xor". It also supports hex litterals, so your code could look something like:

$bcc0 = 0x88 xor $uid[0] xor $uid[1] xor $uid[2]

$bcc1 = $uid[3] xor $uid[4] xor $uid[5] xor $uid[6]

...I think I need to take a shower; I hate PHP

asper said:↑

You need to send this ISO14443A APDU:

1B+4bytes-PWD+2bytes ISO14443A-CRC (7 bytes total).

and you should get 2bytes-PACK back as answer if the command got executed correctly.

I suggest you to find and app that is able to manage ALL the NTAG215 command set (not only ISO14443A standard commands because 1B command is not standard, it is NXP proprietary) or to send the raw command with or without automatically calculating the ISO14443A-CRC.

1) Decrypted the Amiibo dump

2) Read the NTAG215 with Android app NFC TagInfo to get the 7-byte UID.

3) Calculated the UID3 byte.

4) Opened the decrypted Amiibo dump in a hex editor and changed the UID to match the NTAG215

5) Created the keyfile for amiitool

6) Re-encrypted the Amiibo dump with "amiitool -e -k keys.bin -i decrypted.bin -o encrypted.bin"

Now where does this part that you posted come into play? I'm trying to make sure I have all of my ducks in a row so I don't waste any tags here.

#include <iostream>

using namespace std;

int main() {

while (1) {

int UID[4];

UID[3] = 0x88;

cout << "Enter UID0" << endl << "0x";

cin >> hex >> UID[0];

cout << "Enter UID1" << endl << "0x";

cin >> hex >> UID[1];

cout << "Enter UID2" << endl << "0x";

cin >> hex >> UID[2];

for (int i = 0; i < 3; i++) {

UID[3] = UID[3] ^ UID[i];

}

cout << "UID3: " << uppercase << hex << UID[3] << " (0x" << UID[3] << ')' << endl << endl;

system("pause");

}

}

javiMaD said:↑

Now with the correct PACK0 and PACK1 (0x80, 0x80) I get an error 168-0413

I think there is another piece (hmac hash or something) that we are missing. I have checked everything about my clones, PWD, PACK, HMAC at 0x80, settings, etc. and everything checks out but they still don't work. As far as I can tell, 168-0413 is the Wii U equivalent of the 3DS 037-0524 error I have been getting. Has anybody successfully gotten a clone to work?

现在有了正确的pack0和pack1（0x80，0x80）我得到一个错误168-0413

我认为还有一件（HMAC散列或什么的），我们错过了。我检查了一下我的克隆，所有的密码，包，HMAC在0x80，设置，等一切都检查过了但是他们仍然不工作。据我所知，168-0413是Wii U的等效的3DS 037-0524错误我已经得到。有人成功地让克隆人工作了吗？

PokeAcer said:↑

Not at all. 215 has 540 bytes, Amiibo needs 540 bytes, and NTAG215s are 540 bytes. All others are lesser, on in the case with NTAG216s, too much

Actually the problem isn't the size exactly, the problem is that the GET\_VERSION command returns a different value on the NTAG216.

True, but he was using a modified version of amiitool that I'm guessing corrected one of the other signatures. My gut says that it is likely the 0x20 block at 0x34 as that is locked, and isn't used as part of the per amiibo key generation. It could also be the section at 0x60 using a different HMAC key though (like the master one, or another all together).

真的，但他用一个修改后的版本，我猜amiitool纠正另一个签名。我的直觉说那是可能的，锁定0x34 0x20的块，而不是作为每amiibo密钥生成部分。它也可能是一节在0x60使用不同的HMAC密钥虽然（像主，或另一个在一起）。

04237E223A4D81->04237ED1223A4D81D4

There are a number of things you need:

You have to update all 9 bytes of the UID, including the 2 BCC bytes.

You need to update the HMAC signature at 0x80 (amiitool does this for you on reencryption).

You have to re-encrypt the two encrypted segments (this is the main thing amiitool does on reencryption).

You have to properly set the configuration pages on the clone (including the static lock bytes, CC, dynamic lock bytes, CFG0 and CFG1).

You have to properly set the password on the cloned card (based on the UID)

You have to properly set the pack on the cloned card (always the same value)

And something else that we don't know about (my guess is the HMAC at 0x34).

Until we know what that last piece is, it isn't actually possible to create a valid clone.

有很多事情你需要：

你必须更新所有9个字节的UID，包括2 BCC字节。

你需要在0x80更新hmac签名（amiitool这是否为你重新加密）。

你必须重新加密两加密段（这是最主要的事情amiitool对重新加密）。

你必须正确设置配置页面上的克隆（包括静态锁字节，CC，动态锁字节，cfg0和CFG1）。

你必须正确设置密码的克隆卡（基于UID）

您必须正确地在克隆卡上设置包（总是相同的值）

别的，我们不知道（我猜是0x34 HMAC）。

直到我们知道最后一块是什么，实际上不可能创建有效的克隆。

fiveighteen said:↑

So we need to write the first 9 bytes every time, not 8? Easy change to make, but nobody mentioned it before exceptSupercool330 on Page 10. I kind of thought it was done during the encryption or something for some reason. I'll make changes..

In my tests I do the following: ["Step by step" guide]

- Dump Amiibo

- Decrypt Amiibo and save to amiibo\_mod

- Get UID and BCC0/1 from the blank tag

- Put UID1/2/3 + BCC0 + UID4/5/6/7 at offset 0x1D4 (8 bytes)

- Put BCC1 at offset 0x000 (1 byte)

- Put write password at 0x214 (4 byte)

- Put PACK0/1 at 0x218 (2 byte)

- Save file and copy in a new file (amiibo\_base)

Now in amiibo\_base

- Change at 0x002 to 0x00 0x00 (2 bytes), default blank tag LOCK0/1

- Change at 0x208 to 0x00 0x00 0x00 0xBD 0x04 0x00 0x00 0xFF 0x00 0x05 (10 bytes), default blank tag LOCK2-3-4/CFG/MIRROR/AUTH/ACCESS

- Save

- Encrypt amiibo\_mod and amiibo\_base

- Write amiibo\_base to blank tag

- Write amiibo\_mod to blank tag

所以我们需要每次写前9个字节，而不是8个字节？简单的改动，但没有人提到过10页exceptsupercool330。我认为它是在加密过程中完成的，或者出于某种原因。我会改变的。

在我的测试中，我做了以下几点：[一步一步]指南]

转储amiibo

-解密amiibo并保存到amiibo\_mod

从空白标签得到的UID和bcc0 / 1

把uid1 / 2 / 3 + bcc0 + uid4 / 5 / 6 / 7在偏移0x1d4（8字节）

把BCC1偏移0x000（1字节）

放在0x214写密码（4字节）

把pack0 / 1在0x218（2字节）

在一个新的文件保存文件及复印件（amiibo\_base）

现在amiibo\_base

变化在0x002为0x00（2字节），默认的空白标签lock0 / 1

变化在0x208为0x00 0x00 0x00 0x00 0x00 0xbd 0x04 0xff 0x05（10字节），默认的空白标签lock2-3-4 / CFG /镜子/认证/接入

-保存

加密amiibo\_mod和amiibo\_base

写amiibo\_base空白标签

写amiibo\_mod空白标签

Looks good to me now.

7-byte NTAG215 UID: 04C37A52C23E80

BCC0 = 0x88 ^ UID0 ^ UID1 ^ UID2 = 0x88 ^ 0x04 ^ 0xC3 ^ 0x7A = 35

BCC1 = UID3 ^ UID4 ^ UID5 ^ UID6 = 0x52 ^ 0xC2 ^ 0x3E ^ 0x80 = 2E

9-byte UID: 04C37A3552C23E802E

(That last "E" on yours doesn't get capitalized, but the rest of the letters do)

Can someone send me a hash of the amiibo keyfile? I think mine's not correct.:unsure:

Md5 of my keyfile: 2551afc7c8813008819836e9b619f7ed

As an experiment, I scanned a link Amiibo, and restored it to a toad Amiibo (which writes everything except the locked pages from page 0x0D to 0x20), and it still scanned fine. This confirms that whatever the issue is, it is definitely somewhere in the locked area. There are three sections of memory in this locked region:

0x34 - 0x54: A hash according to the doc on 3dbrew. This section is used as the third segment when computing the hash at 0x80.

0x54 - 0x60: We know the first 8 bytes of this section are the Amiibo id that encodes the character. I'm not sure what the last 4 bytes are.

0x60 - 0x80: Probably another hash according to the doc on 3dbrew. This section is used when computing the per Amiibo keys. It is the second half of the base seed.

If these sections are in fact hashes, they must be of only immutable data such as the UID. Besides this section, every part of the user memory is writable, and apart from the UID, all CFG bytes are fixed (identical between Amiibo ). This means we only have a few options:

0x34 - 0x54 is a hash using the per Amiibo hmac key or the master hmac key of some data from 0x54 - 0x80 plus the UID, and potentially some other salt (unlikely; normally the key is the salt).

0x5C - 0x60 is actually the bit we are interested in somehow. I'm still trying to figure out exactly what this is.

0x60 - 0x80 is a hash using the master hmac key of some data from 0x34 - 0x60 plus the UID.

I tried all the obvious hash combinations (my best bet was 0x00-0x08 and then 0x54-0x80 since those are the last two segments used in the hash at 0x80), but couldn't find any matches. If we could figure out what 0x5C - 0x60 (page 23) is, it would be helpful.

Edit: 0x5C - 0x60 doesn't seem to be a good candidate, there is far too little entropy there. Note though that it definitely isn't part of the character ID as two different Amiibos of the same character have different values.

作为一个实验，我扫描链路amiibo，和恢复到一个蛤蟆amiibo（除了锁定页面从页面0x0d去0x20写的，还是扫描的好）。这证实了无论问题是什么，它肯定是在锁定区域的某个地方。这个锁定区域有三段内存：

0x34 - 0x54：根据3dbrew文件哈希。这一部分是作为第三段在计算散列在0x80。

0x54 - 0x60：我们知道本节的前8个字节编码字符的amiibo ID。我不知道最后4个字节是什么。

0x60 - 0x80：或许还有另一个哈希根据3dbrew doc。这部分是用来计算每amiibo键。这是基础种子的下半部分。

如果这些部分是事实上的哈希值，他们必须是不可变的数据如UID。除了这部分用户记忆的每一个部分都是可写的，除了UID，所有字节是固定的（相同的CFG桩在Amiibo）。这意味着我们只有几个选项：

0x54 0x34 -是一个哈希使用每amiibo HMAC密钥或从0x54 - 0x80加UID数据大师HMAC密钥，和潜在的其他一些盐（不可能的；通常的关键是盐）。

0x5c - 0x60实际上是一些我们感兴趣的某种。我还在努力弄清楚这到底是什么。

0x60 - 0x80哈希使用主HMAC密钥的一些数据从0x34 - 0x60加UID。

我尝试了所有明显的散列组合（我最好的赌注是0x00-0x08然后0x54-0x80因为那最后两段0x80哈希中使用），但找不到火柴。如果我们能找出0x5c - 0x60（页23），这将是有帮助的。

编辑：0x5c - 0x60似乎不是一个很好的候选人，有太少的熵有。需要注意的是，它肯定是字符的ID不作为两种不同的理论

I just came to a realization. I was thinking about different sections of memory that could be hashed with the derived per amiibo hmac key, and I realized that you can't actually use that key since part of the seed is the write counter from the amiibo. However, the write counter isn't used with the "locked secret" keyset as the magic is 16 bytes long. This also totally explains why there are two sets of keys, the "unfixed infos" is used for the unfixed parts of the amiibo that can change, and the "locked secret" is used for the locked parts that can only be written once. This also explains why none of my hashing turned up any results; I was using the wrong keyset. I suspect we need to build a seed using the "locked secret" keyset, and then either hash it with the "locked secret" hmac key, take the first 32 bytes from the drbg, or generate a key with the drbg and decrypt/hash something. Regardless, the "locked secret" seed will still use the portion of the amiibo at 0x60, so almost certainly the target block of memory is 0x34-0x54. We just need to try various things until we get something that matches.

我刚刚意识到。我想不同的内存区域，将讨论与派生每amiibo HMAC密钥，我意识到你不能用这把钥匙从种子的部分是写从amiibo计数器。然而，写计数器不是用“锁定键盘为神奇的秘密”是16字节。这也完全解释了为什么有两套钥匙，“无信息”用于的amiibo可以改变非固定部，和“锁定的秘密”是用来锁定部分，只能写一次。这也解释了为什么我的哈希都没有任何结果；我用错了键盘。我想我们需要使用“锁定的秘密”键建立一个种子，然后是哈希与“锁定的秘密“HMAC密钥，从这个带的第一个32字节，或生成密钥的解密/散列东西数。不管怎样，“锁定的秘密”的种子仍然会使用在0x60的amiibo的部分，那么几乎可以肯定的是0x34-0x54目标内存块。我们只需要尝试各种各样的东西，直到我们得到匹配的东西。

mznova said:↑

I think most of us don't care about UI. You can have clickable text on a black screen.:D

I care about UI

I'm not actually comfortable posting code to do this (don't want to get in trouble with anybody, and the previous code I posted is really just a python port of amiitool with some added nfc stuff). The missing piece however is in fact using the locked secret keys to generate another derived keyset, and then using the hmac key from that keyset to hash the last two pieces hashed for the key at 0x80 (0x00 to 0x08 and 0x54 to 0x80). I actually just guessed this, and it was essentially my first guess after I thought to use the locked secret keys. I'll let somebody else write up the final code and share it.

"The missing piece however is in fact using the locked secret keys to generate another derived keyset, and then using the hmac key from that keyset to hash the last two pieces hashed for the key at 0x80 (0x00 to 0x08 and 0x54 to 0x80)"

我不舒服的脚本代码来做这个（不想惹任何人，和以前的代码我贴是真的只是一个Python amiitool港的一些附加NFC的东西）。失踪的一块，然而实际上是使用锁定密钥生成另一个派生密钥集，然后使用HMAC密钥，密钥集哈希散列在最后两块0x80的关键（0x00到0x08和0x54到0x80）。实际上我只是猜到了这一点，在我想使用上锁的密钥之后，这基本上是我的第一个猜测。我会让其他人写最后的代码并分享它。

“失踪的一块，然而实际上是使用锁定密钥生成另一个派生密钥集，然后使用HMAC密钥，密钥集哈希散列在最后两块0x80的关键（0x00到0x08和0x54到0x80）”

Unfortunately I don't really understand this, I have a kind of idea of what this means, but not enough to code anything.

You MAY get a ntag215 BUT it will NOT have "888 useable bytes".

http://www.nxp.com/products/identif...144-504-888-bytes-user-memory:NTAG213\_215\_216

EEPROM

180, 540 or 924 bytes organized in 45, 135 or 231 pages with 4 bytes per page

144, 504 or 888 bytes freely available user Read/Write area (36, 126 or 222 pages)

4 bytes initialized capability container with one time programmable access bits

Field programmable read-only locking function per page for the first 16 pages

Field programmable read-only locking function above the first 16 pages per double page for NTAG213 or per 16 pages for NTAG215 and NTAG216

Configurable password protection with optional limit of unsuccessful attempts

Anti-tearing support for capability container (CC) and lock bits

ECC supported originality check

Data retention time of 10 years

Write endurance 100,000 cycles

Ok, back on track.

We still have to know more about the amiibo data structure.This is the stuffwe know at the moment.@Supercool330said this a few days ago:

Supercool330 said:↑

I updated the layout on the Wikitemp Amiibo page to document the entire layout. I'll add descriptions for each byte when I have some time. There is really only two things I haven't figured out. The first is what the data on page 23 is used for. This is the page immediately after the character data pages, and as far as I can tell it isn't really used for anything currently. This means that if we are doing something wrong on this page (which is locked) it is possible that this could be used to detect a fake. If people could post UIDs along with pages 21 through 23, that would be great. The second thing is exactly how the 0x20 byte block at page 24 is generated. You can use whatever values you want here and the tag seems to work fine, so my assumption is that it is random (which makes sense as it is essentially used as salt for the DRBG used to generate per amiibo keys). However, if it is a hash of something, this could again be used to detect a fake. In fact, generating a valid (but corrupt) Amiibo dump is super easy, you can just set every byte to random, and then write the 2 pages with the character data.

@Supercool330,take this. It's a lot of scanned amiibos info, it could be helpful to figure out the page 23.

Also, we have to still to figure out this:

Supercool330 said:↑

The missing piece however is in fact using the locked secret keys to generate another derived keyset, and then using the hmac key from that keyset to hash the last two pieces hashed for the key at 0x80 (0x00 to 0x08 and 0x54 to 0x80).

We will assume thatthe app@Skyforce77modded works, and can write bin files on blank tags. We can't test yet if the tags it writes work, but at least the interface does so. It would be great if him can fix his app to work on <5.0 devices. (It crashes when it tries to open the file browser on <5.0 devices)

nurofen,PecrowandNightwishlike this.

好，回到正轨。

我们还需要知道更多关于amiibo数据结构，这是目前知道的stuffwe。@ supercool330said这前几天：

supercool330说：↑

我更新的wikitemp amiibo页面布局文件的整个布局。当我有时间的时候，我会给每个字节添加描述。只有两件事我还没弄明白。第一个是第23页上的数据用于。这是字符数据页面之后的页面，据我所知，它目前还没有真正用于任何东西。这意味着如果我们在这个页面上做了一些错误（这是锁定的），可能会被用来检测一个假货。如果人们能够后随着21页23流体通过，这将是伟大的。第二件事是如何在24页的0x20字节块生成。你可以在这里和标签似乎工作好用什么样的价值观，所以我认为这是随机的（这是因为它本质上是作为用于生成每amiibo键DRBG盐）。然而，如果它是一个散列的东西，这可以再次被用来检测假货。事实上，产生一个有效的（但腐败）amiibo转储是超级简单，你可以设置每一个字节的随机，然后写出2页的字符数据。

“supercool330，借此。这是一个很大的扫描amiibos信息，这将有助于找出23页。

此外，我们还得弄清楚这一点：

supercool330说：↑

失踪的一块，然而实际上是使用锁定密钥生成另一个派生密钥集，然后使用HMAC密钥，密钥集哈希散列在最后两块0x80的关键（0x00到0x08和0x54到0x80）。

我们将假定应用程序”skyforce77modded作品，并可以在空白标签写入bin文件。我们不能测试它写的标签是否有效，但至少界面是这样的。如果他能修复他的应用程序以使用< 5设备的话，那就太好了。（当它试图打开< 5设备上的文件浏览器）时就会崩溃。

pecrowandnightwishlike Nurofen，这。

nurofen said: ↑

I am stuck again, I think I am getting closer, but I want to check my keys file (The locked secret one, not the unfixed infos) can someone help me out here, maybe an md5 checksum or something?

I posted the md5 checksum of the keyfile in this thread:

fraret said: ↑

Md5 of my keyfile: 2551afc7c8813008819836e9b619f7ed

nurofen said:↑

As I say, I am pretty close. I think I understand, but really need a few more clues / help from@Supercool330as he has got this to work.

Basically I understand using the 'unfixed infos' key and hashing against 0x011:0x034,0x0A0:0x208,0x034:0x054,0x000:0x008 and 0x054:0x080 to produce the Unfixed HASH for the data at 0x80:0xA0

Now what I think we need before we do this is create the 'Locked secret' HASH at 0x034:0x054. This is where I am stuck, I know we use the 'Locked secret' keyset but I can't work out which areas to HASH against , my guess would be the areas that are not updateable, i.e. 0x208:0x21c ,0x000:0x008 and 0x054:0x080.

As the area 0x034:0x054 is not encrypted I should be able to check the generated data against the actual data. However I am not having much luck. It could be that my keyfile is incorrect.

If anyone can give us some more clues that would be great.

Using 'locked secret' keyset.

现在我想在我们做我们所需要的是在0x034:0x054创建“锁定秘密散列。这是我在哪里卡住了，我知道我们的“秘密”键盘锁但我不能找出哪些地区散列的反对，我想可能是地区不可更新的，即0x208:0x21c，0x000:0x008和0x054:0x080。

作为地区0x034:0x054没有加密，我应该能够检查所生成的数据与实际数据。但是我运气不好。这可能是我的密钥文件不正确。

如果有人能给我们更多的线索，那将是伟大的。

使用“秘密”键盘锁。

'tag' format:

Calc hash of (0x000:0x007 + 0x054:0x07F) (52 bytes), put this hash (32 bytes) at 0x034

'internal' format:

Calc hash of (0x1D4:0x207) (52 bytes), put this hash (32 bytes) at 0x1B4

SHA1 please

nurofen said:↑

Thanks, I don't suppose you can give the md5 of the locked secret file?

MD5 0ad86557c7ba9e75c79a7b43bb466333

SHA1 ad676ac04c6e7861924093654bd67ff4807ebc53

HiddenRambler,oldsk00l,Melon\_\_Breadand1 other personlike this.

Code:

0C 0D 0E 0F

b6 a3 c2 05

74 00 00 10

f2 cf d2 9b

96 0f ae d4

45 05 47 66

Is your new keyset ok?

MD5 45fd53569f5765eef9c337bd5172f937

SHA1 bbdbb49a917d14f7a997d327ba40d40c39e606ce

Is your new keyset ok?

It is, hashes match. Kind of out of ideas at the moment. What did you write the data with, if you don't mind me asking?

Well, seems like I missed somethimg important!

Can someone clear what is meant by "new keyset". My "old keyset" is SHA-1: 125083246346D7FB76E198D759CF1167E503A662. What do I need for the "new keyset"?

$ md5sum ~/javimadgit\_amiitool/amiitool/all\_in\_one\_keys.bin

45fd53569f5765eef9c337bd5172f937 ~/javimadgit\_amiitool/amiitool/all\_in\_one\_keys.bin

$ sha1sum ~/javimadgit\_amiitool/amiitool/all\_in\_one\_keys.bin

bbdbb49a917d14f7a997d327ba40d40c39e606ce ~/javimadgit\_amiitool/amiitool/all\_in\_one\_keys.bin

To find the key file Google:Use this to Encrypt your custom Amiibo NTag pastebin

You'll find a .bin file, open in HxD and "split" it in two parts (the first 5 rows and the second 5 rows), then save the first part to "unfixed key.bin" and the second to "locked key.bin". Put the three .bin files (unfixed, key and amiibo) into your Android Phone, boot TagMo2, place a NFC tag under the phone and write the new amiibo

sha256hmac(tagKeys.hmacKey, sizeof(tagKeys.hmacKey), plain + 0x1D4, 0x34, \*\*\*\*\*cipher\*\*\*\*\* + HMAC\_POS\_TAG);

sha256hmac(dataKeys.hmacKey, sizeof(dataKeys.hmacKey), \*\*\*\*\*\*plain\*\*\*\*\* + 0x029, 0x1DF, cipher + HMAC\_POS\_DATA);

the tag checksum is used by the data checksum. however since the plain data is never updated the data checksum still uses the invalid checksum.

if you look at@javiMaD's original code he does it correctly. Also@javiMaDyour latest code is also wrong due to merging the changes from upstream with this bug.

tagmo2.1 work perfectly

but after flash it isn't possible to format because page 13 to 31 is read only:(

if you want try with blank tag i hope to made this with a good dump

if you look lock byte lock0 lock1 lock2-lock4 you can see a value of defaut dump file but if you want format blank tag after a first flash you must write this byte to 00 value(default value of blank tag)

Hi,

i'm very new to this topic of amiibo spoofing. I've read a lot and got some NTAG215 to work with help of TagMo. Still i'm not sure if it makes sense to get a n2elite. I'm looking for the "safest" method that has the lowest possibility to get detected in future, because i'm afraid of getting my nintendo account flagged for cheating or get suspended or something else. If read of the current problem with allmost all amiibo dumbs beeing 540 bytes and not 572 bytes which could cause trouble in future. So my primary questions are the following:

我很新的这一话题的amiibo欺骗。我读了很多，有一些ntag215与帮助tagmo工作。我仍然不确定它是否有意义有n2elite。我在找有检测未来的可能性最低的“安全”的方法，因为我怕我的任天堂帐户标记为作弊或被暂停或别的东西。如果当前的问题几乎都只能读540个字节，而不是作为Amiibo 572字节可日后造成麻烦。所以我的主要问题如下：

Is it possible to store an full 572b dump to an ntag215 which has only a size of 540b?

How can a spoof of an unreleased amiibo (like the majoras mask link) can be an additional risk? Is it the bin file itself thats the threat or just the fact that the amiibo isnt released yet?

Is it possible to detect the n2 elite as a non-amiibo because it is not a regular nfc tag like the ntag215?

All in all: which method is safer n2 elite or ntag215?

I love to support nintendo but in case of botw it's just rediculous... dont plan to spend so much money for figures that i didnt want to be in my living room only for ingame-items that i cant get another way. If i want an amiibo for the figure i will buy it anyway like the oncoming cloud amiibo:)

Looking forward to your answers

...Still i'm not sure if it makes sense to get a n2elite. I'm looking for the "safest" method that has the lowest possibility to get detected in future, because i'm afraid of getting my nintendo account flagged for cheating or get suspended or something else. If read of the current problem with allmost all amiibo dumbs beeing 540 bytes and not 572 bytes which could cause trouble in future....

Is it possible to store an full 572b dump to an ntag215 which has only a size of 540b?

Is it possible to detect the n2 elite as a non-amiibo because it is not a regular nfc tag like the ntag215?

These two questions are related. The NTAG215 has a signature that authenticates that the card was made by the manufacturer that it claims to be (like NXP.) Those 32 extra bytes are the signature. An NTAG215 (whether it's blank, or an amiibo, or on a shelf) either already has a valid signature, or it's a knockoff. As long as your NTAG215 was legitimately manufactured, you're good. If not, there's nothing you can do about it except to get your tags from a reliable vendor.

…我还是不确定它是否有意义有n2elite。我在找有检测未来的可能性最低的“安全”的方法，因为我怕我的任天堂帐户标记为作弊或被暂停或别的东西。如果当前的问题几乎都只能读540个字节，而不是作为Amiibo 572字节，可能导致在未来的…麻烦

可以存储一个完整的572b转储到一个ntag215只有大小540b？

它是可能检测N2精英作为非amiibo因为它不是一个普通的NFC标签像ntag215？

这两个问题是相关的。的ntag215有签名认证的卡是由制造商，它声称是（像NXP。）这32个额外的字节的签名。一个ntag215（不管它是空白的，或者一个amiibo，或在一个架子上）已经有一个有效的签名，或是仿制品。只要你的ntag215是合法生产的，你很好。如果没有，你除了能从可靠的供应商那里得到你的标签，你什么也做不了。

n2 wants 572 byte dumps so they canemulatethe same signature that therealsource NTAG215 had. Without it, the signature check would fail -IFNintendo ever decided to check it.

TL;DR 572 vs 540 is a problem for tag emulation, not tag copying.

How can a spoof of an unreleased amiibo (like the majoras mask link) can be an additional risk? Is it the bin file itself thats the threat or just the fact that the amiibo isnt released yet?

The risk with unreleased amiibos is that since the amiibo isn't released, any one who is using it (unless Nintendo gave to them for testing) is cheating. In an online game/system, that would be pretty easy to detect. In a completely offline situation, it's not as much of a risk.

All in all: which method is safer n2 elite or ntag215?

Don't let people make choices for you. Just learn how things work, and draw your own conclusions. In regard to the signature matter, if I were Nintendo, and the signatures were affected by serials, I'd disable serials that I saw being used excessively. A move like that could cause problems for n2b

热释光；540 572 VS博士是一个问题emulation带键槽，以复制。

嘿，恶搞怎么能一个unreleased amiibo（majoras渣样的链路可以是额外的风险吗？它是在本文件本身或威胁，这是真的，这是amiibo发布的，不是吗？

风险与unreleased amiibos是从《amiibo没有发布任何一，谁使用它（除非任天堂的礼物到冰试验）的欺诈。在一个在线游戏系统，这将是很容易检测两种。在一个完全脱机的情况下，它不为太多的风险。

所有的好方法：冰，N2或ntag215精英更安全吗？

不要轻易让你选择的人。只是学习如何工作的事情，和绘制你自己的结论。在这两个方面的事，如果我是任天堂和签名是受影响的城市。在disable serials D，看到excessively被使用。A我喜欢搬家，因为n2b问题