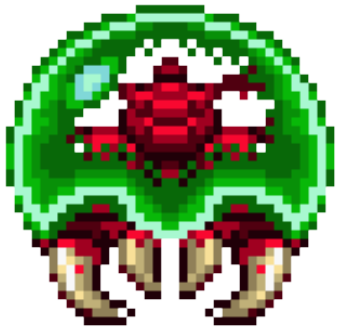
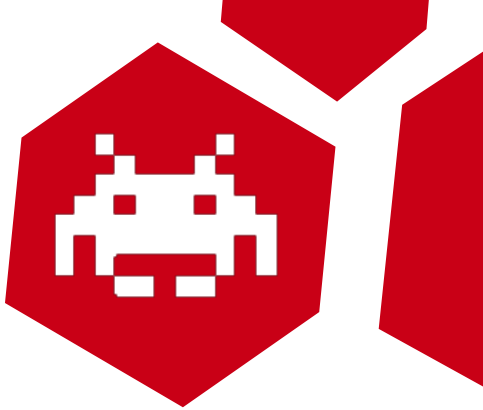


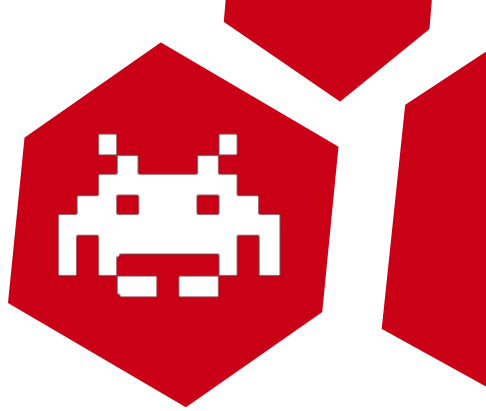
Don't Give Credit: Hacking Arcade Machines

Who am I?



- Ronald Huizer
  - Senior Security Researcher, Immunity, Inc.
  - ronald@immunityinc.com
  - I enjoy computer science, toying with hardware, go, a whole lot of japanese cartoons and computer games.

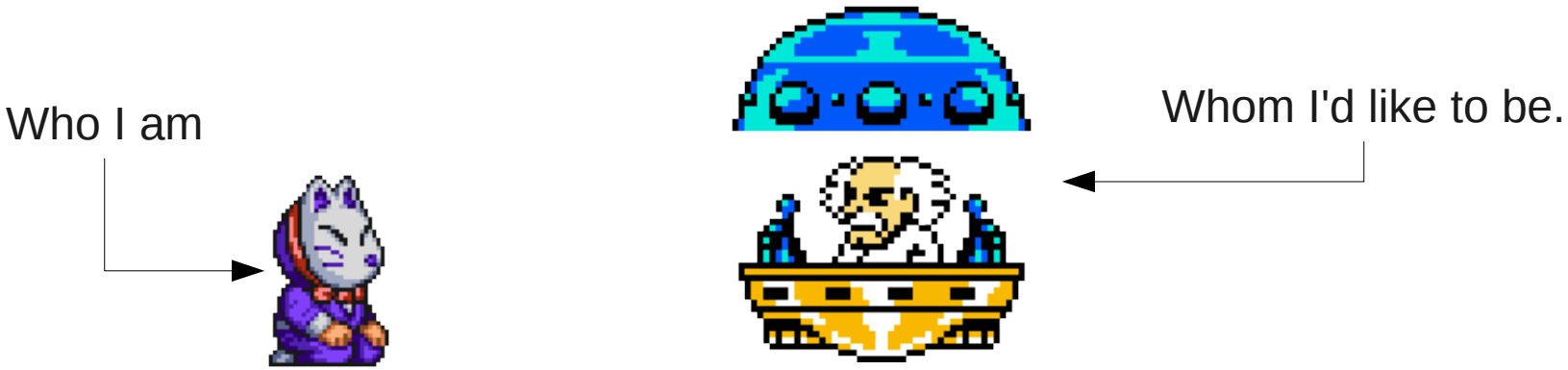
Who am I?



- Ronald Huizer
  - Senior Security Researcher, Immunity, Inc.
  - ronald@immunityinc.com
  - I enjoy computer science, toying with hardware, go, a whole lot of japanese cartoons and computer games.

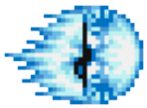
Who am I?

- Ronald Huizer
  - Senior Security Researcher, Immunity, Inc.
  - ronald@immunityinc.com
  - I enjoy computer science, toying with hardware, go, a whole lot of japanese cartoons and computer games.

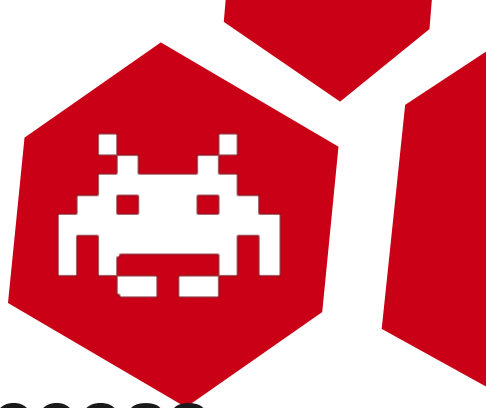


Attacking Arcade Machines

- Why attack arcade machines?
- Fun and free plays.
- Not so much profit, unless you play a lot.
- Living one of my childhood dreams.
- Both the vulnerability and the talk are quite simple.
- This is meant to be fun and practical.



Attack Surface (1)



- Almost all attacks will need physical access.
- We need to make a distinction
  - Obvious attacks such as opening the machine, or attaching odd peripherals and rebooting it.
  - Non-obvious attacks that resemble normal use. These are probably impossible on many older arcade machines.

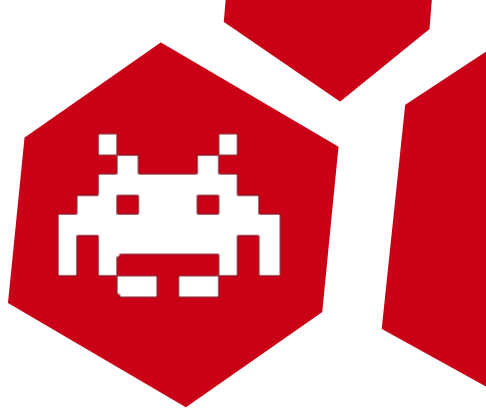
Attack Surface (2)

- The obvious attacks won't work, as we'll get kicked out of the arcade or worse.
- We want to be less conspicuous than this:

Attack Surface (3)

- Modern arcade machines often allow for transferable profiles stored on portable devices.
  - Magnetic cards
  - Konami e-AMUSEMENT smart card
  - USB dongles
  - Probably more schemes, especially in Japan.
- This gives us more attack surface using either malicious hardware devices, or by malicious data on official devices.

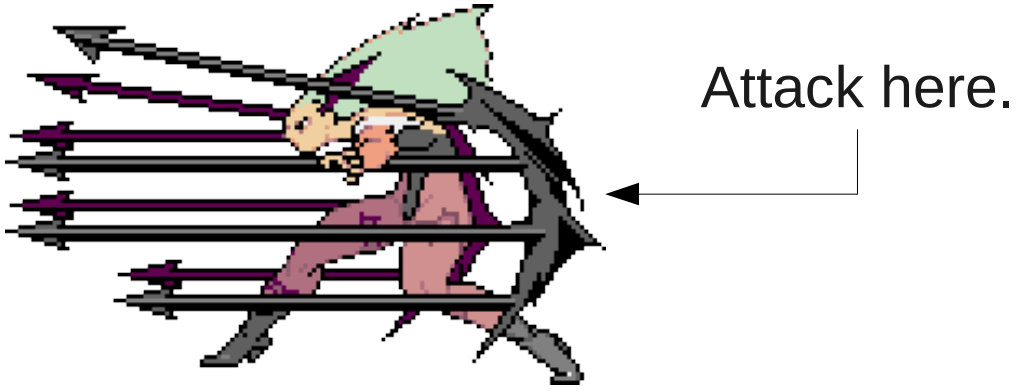
Attack Surface (4)



- We pick the easiest attack surface.
- Consider game profiles loaded from and stored to USB dongle.
  - If profile handling is done wrong, we can simply insert a USB dongle with malicious payload.
  - Very covert: inserting a dongle is a common task performed by many players, and won't attract unwanted attention.

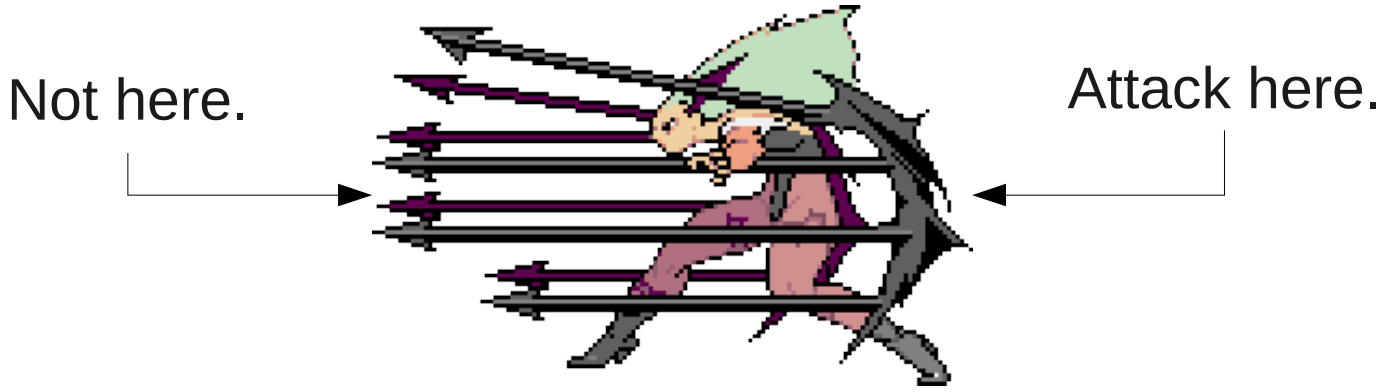
Attack Surface (4)

- We pick the easiest attack surface.
- Consider game profiles loaded from and stored to USB dongle.
  - If profile handling is done wrong, we can simply insert a USB dongle with malicious payload.
  - Very covert: inserting a dongle is a common task performed by many players, and won't attract unwanted attention.



Attack Surface (4)

- We pick the easiest attack surface.
- Consider game profiles loaded from and stored to USB dongle.
  - If profile handling is done wrong, we can simply insert a USB dongle with malicious payload.
  - Very covert: inserting a dongle is a common task performed by many players, and won't attract unwanted attention.



What are we attacking?

- In The Groove 2
- Dancing simulator made by RoXoR games.
- Uses USB dongles to store profiles.

What are we attacking?

- In The Groove 2
- Dancing simulator made by RoXoR games.
- Uses USB dongles to store profiles.
- Allows geeks to dance like Michael Jackson.

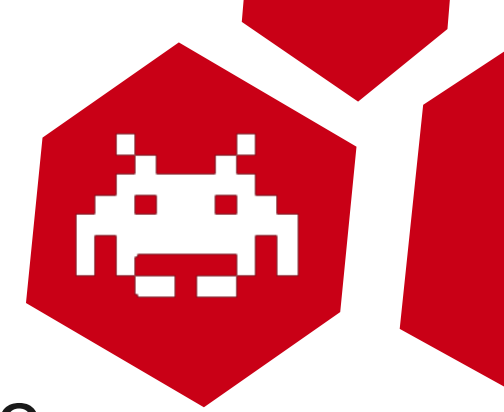
What do we know? (1)

- There is a PC as well as an arcade version.
  - We'll use ITG2PC and ITG2AC for these versions.
  - We can tinker with the PC version easily and test our ideas.
  - After testing them on ITG2PC, we try ITG2AC.
- ITG2AC is running on x86-32 Linux.
  - Most of us will be in our comfort zone.

What do we know? (2)

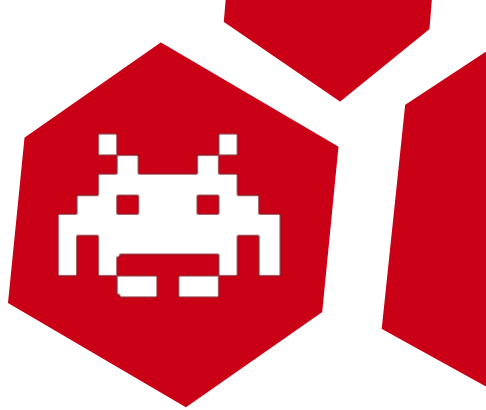
- ITG2 software based on a modified version of StepMania, an open source dancing simulator.
  - Allows for easier reverse engineering.
- There is an open source project dedicated to reimplementing the game.
  - OpenITG did an excellent job at reversing and reimplementing parts of the game.

What is on the USB stick?



- Edits of existing songs on the machine.
- Custom songs (needs to be enabled).
- Signed screenshots (to prove scores).
- Signed score profile and backups.
  - Stats.xml / Stats.xml.sig / DontShare.sig
- Song catalogues, preferences, etc.
- ITG2AC and ITG2PC sticks are not portable
  - Because the signing keys differ.

Stats.xml: user profile data



- XML formatted file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/xsl" href="Stats.xsl"?>

<Stats>
<CalorieData>
<CaloriesBurned Date='2005-02-26'
>468.587524</CaloriesBurned>
</CalorieData>
<CategoryScores/>
...
<Data>
local tab1 = { }
return tab1
</Data>
...
```

Stats.xml: user profile data

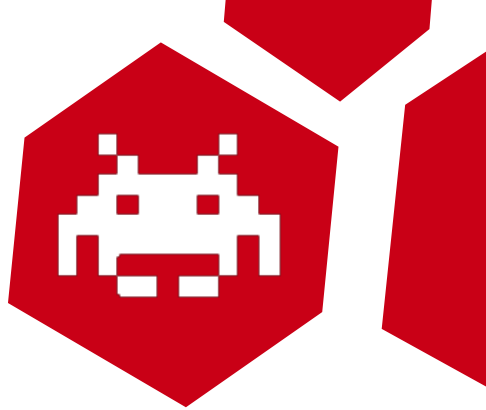
- XML formatted file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/xsl" href="Stats.xsl"?>

<Stats>
<CalorieData>
<CaloriesBurned Date='2005-02-26'
>468.587524</CaloriesBurned>
</CalorieData>
<CategoryScores/>
...
<Data>
local tab1 = { }
return tab1
</Data>
...
```

What reading XML does to people.

XML parser flaws



- XNode::LoadAttributes() has issues.
- It will scan past 0-byte if there is a double or single quote character before it.
- tcsskip() and tcsechr() are scary, as they *always* return a non-NULL pointer.
- Lots of over-indexed reads, but hard to find over-indexed writes.
- Need a better bug.

XML parser flaws

- XNode::LoadAttributes() has issues.
- It will scan past 0-byte if there is a double or single quote character before it.
- tcsskip() and tcsechr() are scary, as they *always* return a non-NULL pointer.
- Lots of over-indexed reads, but hard to find over-indexed writes.
- Need a better bug.

This is not a good bug.

User profile loading flaws (1)

- Profile::LoadGeneralDataFromNode() reads XML data from the XML tree, and deserializes.
- Lot of uninteresting numeric and string entries.
- The <Data> tag seems interesting, as it contains embedded LUA data.
- It is only handled for IsMachine() profiles, which are stored on the arcade machine itself.

User profile loading flaws (1)

- Profile::LoadGeneralDataFromNode() reads XML data from the XML tree, and deserializes.
- Lot of uninteresting numeric and string entries.
- The <Data> tag seems interesting, as it contains embedded LUA data.
- It is only handled for IsMachine() profiles, which are stored on the arcade machine itself.
- Are they really?

User profile loading flaws (2)

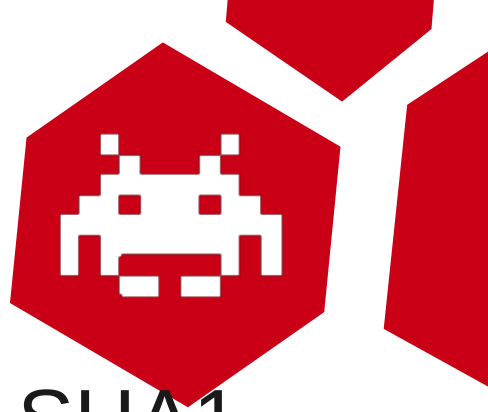
- In OpenITG there is an IsMachine() check.
- Not so in R21 and R23!

```
v29 = GetChildValue(a3, "Data");
if ( v29 )
{
  string_constructor(v29, &sData);
  LoadFromString(a2 + 5000, (int)&sData);
  if ( GetLuaType(a2 + 5000) != LUA_TTABLE )
  {
    Warn((int)LOG, "Profile data did not evaluate to a table");
    sub_84C3C80(*(_DWORD *)LuaHelpers);
    sub_81C2870(a2 + 5000);
  }
}
```

Creating a rogue profile

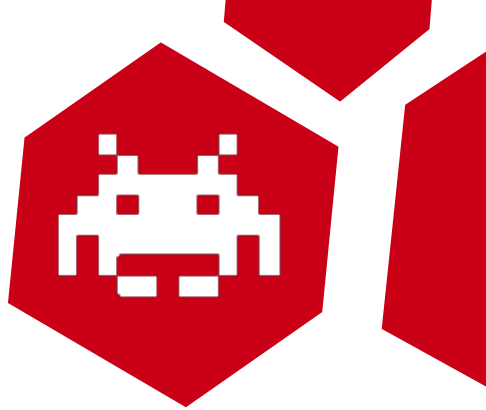
- We have found a way to inject LUA code.
- There's still more work to be done:
  - Signing profiles with malicious LUA code.
    - This requires the signing keys.
  - Finding out what LUA code we can use.
    - Is there a LUA sandbox?
    - Can we escalate to root on the machine?
    - Do we actually need to? What can we do otherwise?

Signing profiles (1)



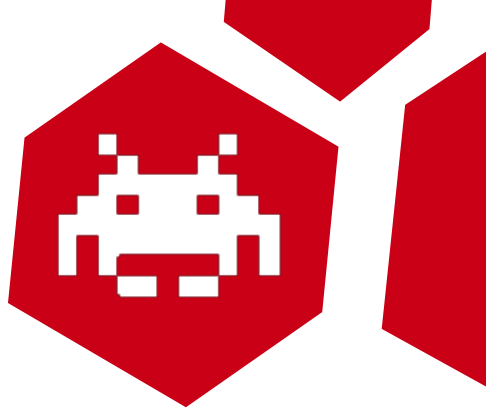
- Profile signing is done using RSA and SHA1.
- Original implementation using crypto++.
- Signing: S(k-, p) = E(k-, h(p))
- Verification: D(k+, S(k-, p)) should be h(p).
- Reimplemented this using OpenSSL, as crypto++ is complicated to use.
- Command line OpenSSL also works.

Signing profiles (2)



- What is signed?
  - Stats.xml with the result in Stats.xml.sig
  - Stats.xml.sig with the result in DontShare.sig
- This double signature is done so people can share verified (machine signed) scores, without their profile being copied.
- You would share Stats.xml and Stats.xml.sig but not DontShare.sig

Signing profiles (3)



- We obviously want the private key.
- ITG2 signs profiles every time someone plays.
- Private key needs to be known to the program.
- Profiles need to be transferable.
  - So the signing keys are shared!
- No revocation scheme in place.
  - Once we leak one key, we're set!

OpenSSL signing / verifying

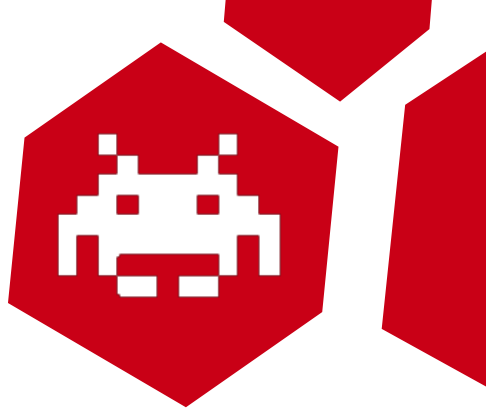
- openssl dgst -keyform DER -sign private.rsa -out Stats.xml.sig Stats.xml

  openssl dgst -keyform DER -sign private.rsa -out DontShare.sig Stats.xml.sig

- openssl dgst -keyform DER -verify public.rsa -signature DontShare.sig Stats.xml.sig

  openssl dgst -keyform DER -verify public.rsa -signature Stats.xml.sig Stats.xml

OpenSSL DER to PEM



- Private key is in PKCS8 DER form.

  openssl pkcs8 -in private.rsa -inform DER -outform PEM -out private.pem -nocrypt

- Public key is in RSA DER form.

  openssl rsa -in public.rsa -inform DER -pubin -pubout -outform PEM -out public.pem

ITG2PC

- The private keys are simply installed.
- They obviously differ from the ITG2AC keys.
- Look for the *.rsa files.
- They come in PKCS #1 / PKCS #8 forms.

A key!

ITG2AC

- Dumping the private keys more complicated.
- We need to crack open the machine first.
  - Attach USB keyboard and Linux disk.
  - Rebooting the machine.
  - Enter + configure BIOS to boot from disk.
  - Mount the ITG2 XFS filesystem and have at it.
  - Will not work on R23, as it rewrites the BIOS password using nvram.ko

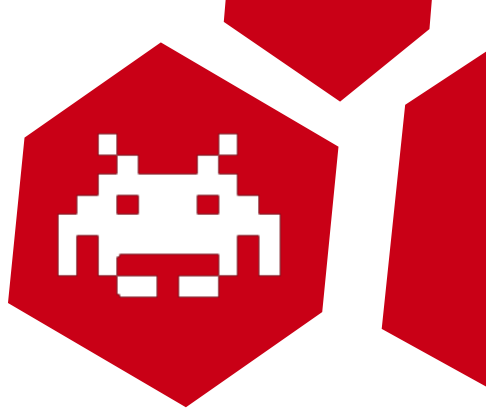
ITG2AC (2)

- We were unable to find the keys on disk.
- /itgdata contains several crypted blobs: data0.zip through data4.zip and patch.zip.
- The keys are most likely in there, as well as the songs and so on.
- We need a way to decrypt those files.

ITG2AC file encryption

- The core algorithm uses SHA-512 and AES-192 in CBC mode.
- The AES keys are managed in two ways.
  - Patch files use a static key, probably because it is easier to deliver patches.
  - The core data files all have unique keys, which differ on all arcade machines. These are managed by a hardware security dongle.

Encrypted file header (1)



```
struct itg2_file_header
{
        char            magic[2];
        uint32_t        file_size;
        uint32_t        subkey_size;
        uint8_t         *subkey;
        uint8_t         verify_block[16];
};
```

Encrypted file header (2)

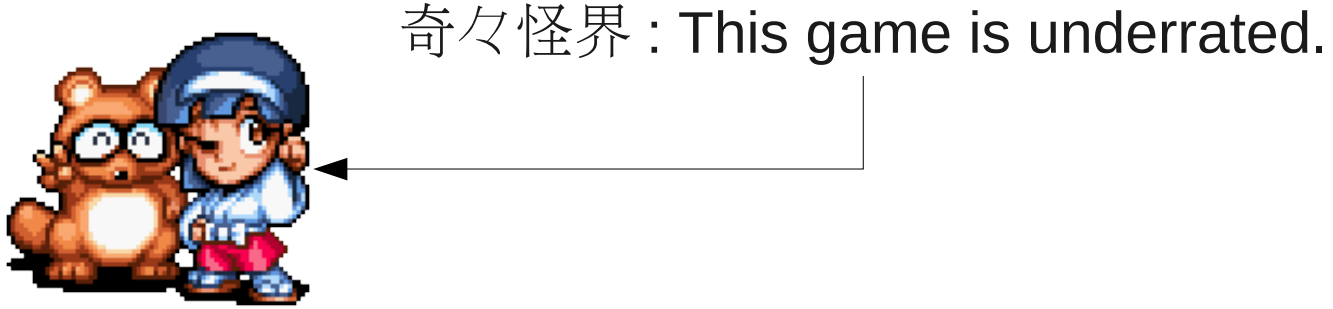
- Magic will be :| for data files and 8O for patch files.
- file_size is the size of the decrypted file, so that padding to blocksize can be ignored.
- subkey_size is the size of the subkey.
- subkey is the size of subkey data.
- verify_block is a block of encrypted static data to determine if a valid key was provided.

File decryption algorithm (1)

- AES-192 keying is used. How these keys are derived we will see later.
- Remember that AES works on 16 byte blocks.
- File is partitioned in blocks of 255 AES blocks.
- Each of these blocks is encrypted using AES in CBC mode.
- The IV is manipulated before every encryption, by subtracting 0 through 16 from IV elements.

File decryption algorithm (2)

- Why does it work like this?
- CBC mode is quirky for file encryption.
- If we encrypt the full file in CBC mode, a single corruption in the worst case will ruin the entire file.
- When partitioning in blocks a single corruption in the worst case ruins the block.



File decryption algorithm (3)

- We get IV repetition per block of 255 blocks. This is a slight weakness, but not fatal for CBC.
- Why they modify the IV is unclear to me.
- It causes some additional confusion, and it does not introduce additional duplicates, so it is probably alright.

AES key recovery (1)

- The AES key for patch files is created running a function similar to SHA512-HMAC.
- It is not a real HMAC, as there is no ipad/opad or key compression performed, but simply does: SHA512(m || k)
- The message is the subkey from the file header.
- The key can be recovered by reverse engineering (or reading the OpenITG code).

AES key recovery (2)

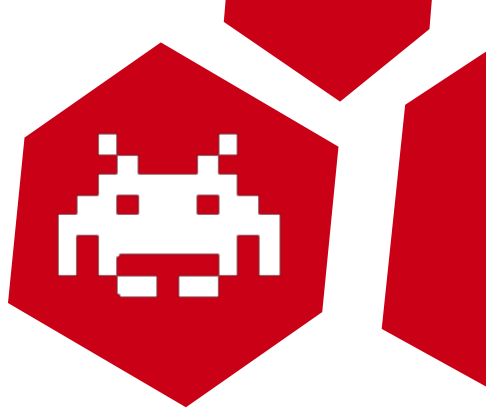
- The AES keys to the data files are stored on an security dongle.
- The dongle is an iButton DS1963S which is used as a SHA-512 HMAC co-processor to deliver the AES keys.
- We don't need the DS1963S secret keys: we can recover the AES key for specific data files.

Fu fu fu, enough crypto already.

DS1963S architecture

- The dongle is connected to the RS232 port of the machine.
- It communicates through a bus protocol called 1-Wire so that the master can communicate with multiple slaves.
- There is a public domain kit available to communicate with the dongle.

DS1963S memory

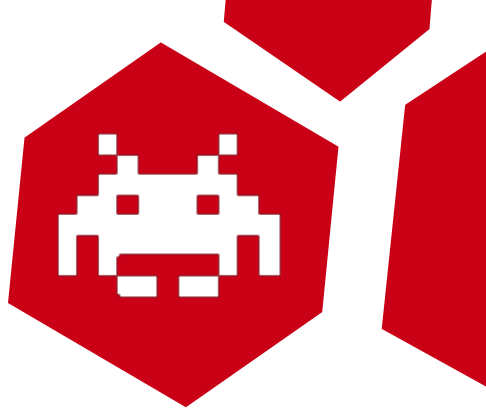


- There are 16 256-bit data pages.
- There are 2 pages holding 4 64-bit secrets each. These are writable, but not readable.
  - Reading the secret pages would break DS1963S security, but we do not need to do this for decrypting the data files.
- There is a 256-bit scratch pad used for reliable transfers from master to slave memory.

DS1963S registers

- TA1 and TA2 hold the LSB and MSB of the target address used in many operations.
- E/S is a read-only counter and status register
  - Bits[0..4]: The ending offset; it holds the last offset into the scratch pad that was written to.
  - Bits[5]: The partial flag (PF); set to 1 when the bits sent by the master are not a multiple of 8.
  - Bits[6]: Unused; should be 0.
  - Bits[7]: Authorization Accepted (AA); set to 1 when the scratchpad has been copied to memory.

DS1963S reliable write (1)



- [0xC3] [TA1] [TA2]

  Erase the scratchpad, filling it with 0xFF. TA is ignored. Clear HIDE flag.

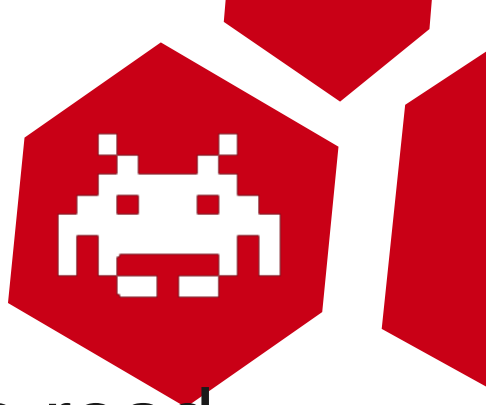
- [0x0F] [TA1] [TA2] [DATA ...] [CRC16]

  Write data to the scratchpad, from the byte offset to the ending offset. If the ending offset is 0x1F, the slave sends back the CRC16 of data read.

- [0xAA]

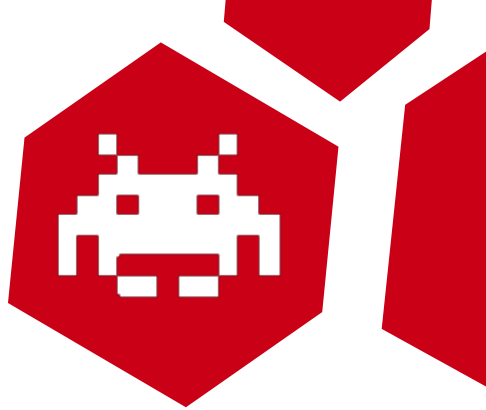
  Read scratchpad. Slave sends back the byte offset, the ending offset, and the scratchpad area for those, and ~CRC16.

DS1963S reliable write (2)



- Comparing the data written to the data read guarantees (almost) no distortions.
- From scratchpad we can then write into data pages and secrets pages.
- All this is performed by the public domain API function WriteDataPageSHA18().

DS1963S SHA functions



- There are multiple SHA functions.
- We will only look at the one relevant to ITG2AC.
- [0x33] [0xC3] SHA-1 sign data.
  - Signs data page 0 or 8 with the secret number 0 or 8, and data from the scratchpad.
  - This is used to generate the AES key from the subkey data in the file header.

DS1963S security (1)

- Secret page security demonstrated broken by Christian Brandt at CCC 2010 through faulting.
- Using real crypto does not make devices secure.

DS1963S security (1)

- Secret page security demonstrated broken by Christian Brandt at CCC 2010 through faulting.
- Using real crypto does not make devices secure.

Would you rather attack SHA-1?

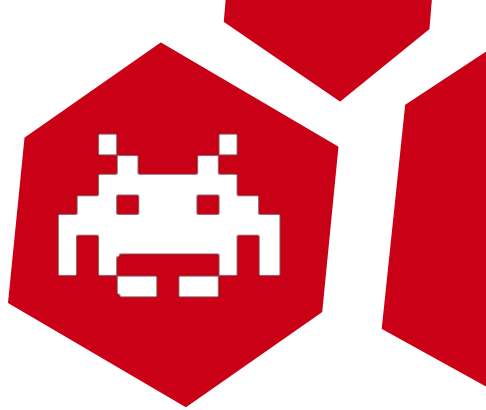
DS1963S security (1)

- Secret page security demonstrated broken by Christian Brandt at CCC 2010 through faulting.
- Using real crypto does not make devices secure.

Would you rather attack SHA-1?

Or the DS1963S protocols?

DS1963S security (2)



- An **untested** idea to dump secrets.
  - The scratchpad and memory do not have to be written in 32-byte blocks.
    - We can write smaller quantities, like 1 or 2 bytes.
  - The Copy Scratchpad command can write secret pages directly.
    - We just can't read secret pages.
  - Partial secret overwrite may be possible?
    - Use Sign data page (SDP) with original secret.
    - Now overwrite 1 byte, and SDP again until correct byte has been found.
    - Repeat: complexity now O(256*8) instead of O(256**8).

DS1963S demonstration

DS1963S demonstration

This octopus is funnier than Cthulhu.

File decryption

- We can now use the DS1963S keys to decrypt the encrypted files.
- This opens the door for unauthorized copying of the game content...
  - Keep in mind that ITG2PC had no DRM whatsoever, so it is of minimal concern.
- It also allows us to use the original files portably in other projects. Think of OpenITG.

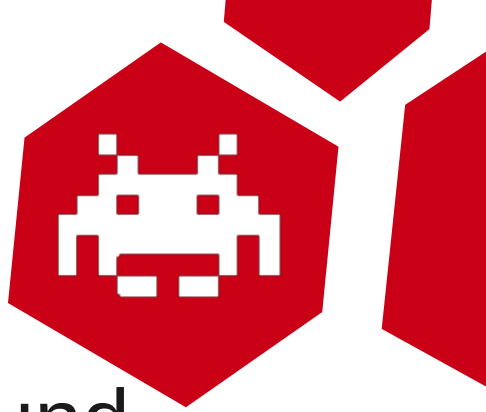
Signing key recovery

- We can now find the profile signing key by decrypting and unpacking data4.zip.
- The keys are in Data/private.rsa and Data/public.rsa.

Using LUA

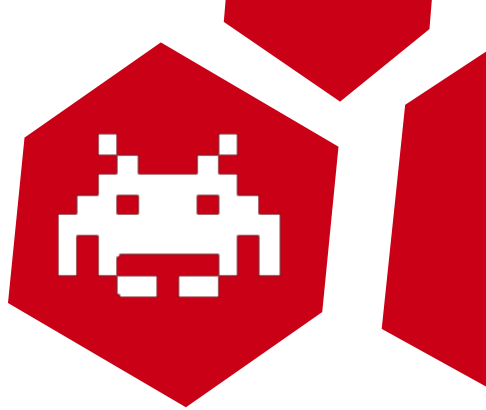
- So we can get LUA code executed by signing profiles with embedded code.
- The LUA environment is sandboxed, there is no support for the os module and so on.
- This means we cannot execute arbitrary code on the machine.
- We can execute the LUA bindings the game provides, and change game state.
- This is what we want anyway really.

LUA game commands



- A brief stepmania reference can be found online at: http://www.stepmania.com/wiki/Lua_scripting_and_Actor_commands
- It differs from the commands in R21, and R23, but there are many similarities.
- GameState.cpp implements ApplyGameCommand() which has some interesting primitives.
- GameCommand.cpp implements these primitives.

LUA game commands (2)



- The one I was looking for as a kid:

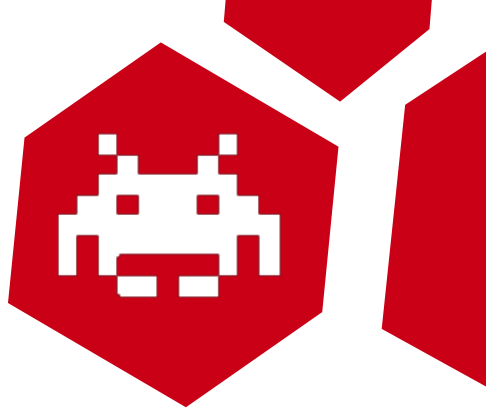
  GAMESTATE:ApplyGameCommand('insertcredit')

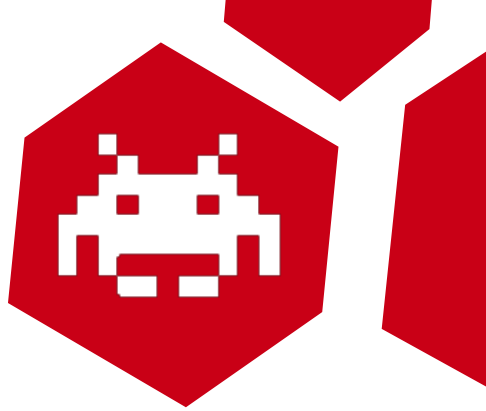
- Signing a profile using this command and using it indeed leads to a free credit.
- The profile loader needs to be invoked, so we need to use one credit to get the rest for free.

Further escalation

- We would need to break the LUA sandbox.
  - We have several flaws, but they are complicated.
  - What more do we want anyway?
    - We can play for free.
    - We can unlock songs.
    - We can transfer scores to the machine.
    - We do not want to mess it up: the sandbox is nice.

Demonstration





Questions? Kupo?