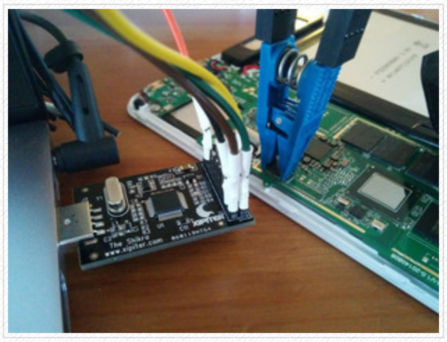
PROGRAMMING THE SHIKRA

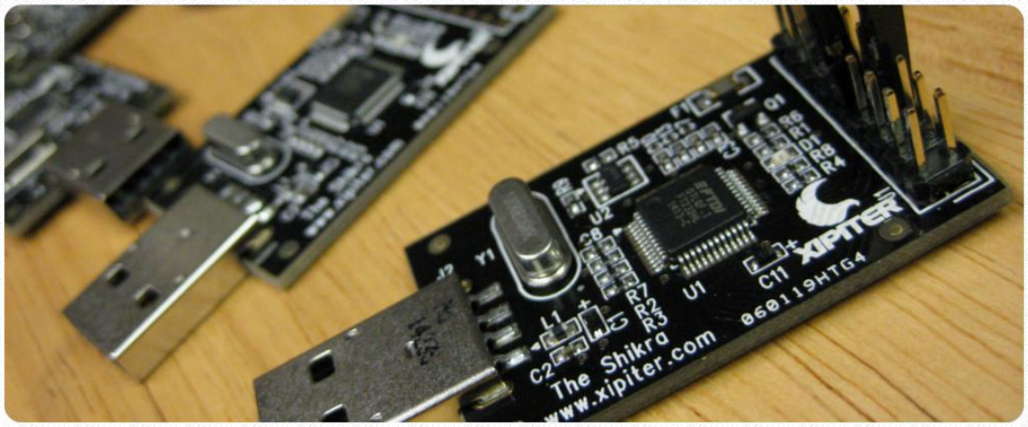
2/1/2016 /

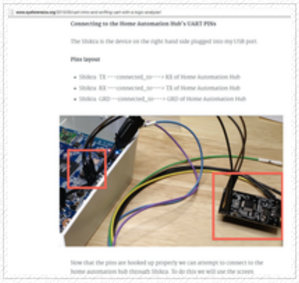

The Shikra saving a "bricked" tablet by invasively re-flashing.

The Shikra is a very powerful tool to have in ones toolbox. (See this blogpost for some of the neat things that can be done with it.) We actually created the device in response to our frustration with tools like the BusPirate, and as a reliable tool for our "Software Exploitation Via Hardware Exploitation" course. See some screenshots of how others have used it on the product page.

The Shikra "just works" out of box. Other than the FTDI drivers, it needs no client-side software. You don't even have to terminal into it (w/ minicom, putty, screen, et al) as you do with the BusPirate (and other tools). This is actually its strength. Because it works entirely in hardware, it is several orders of magnitude faster than tools like the BusPirate for tasks like dumping flash with SPI, extracting firmware with JTAG, etc.

Even though the Shikra works out of box and requires no configuration, we wanted to provide a utility (for those interested in "going deeper" and tweaking the device functionality by programming the onboard EEPROM memory. What follows is a primer in simple binary data manipulation techniques (with Python) and a bit about hardware in general.




Connecting to the Home Automation Hub's UART PINs

The Shikra is the device on the right hand side plugged into my USB port.

Pins layout

- Shikra TX ---connected_to---> RX of Home Automation Hub
- Shikra RX ---connected_to---> TX of Home Automation Hub
- Shikra GRD ---connected_to---> GRD of Home Automation Hub

Now that the pins are hooked up properly we can attempt to connect to the home automation hub through Shikra. To do this we will use the screen

We play "spot the Shikra" frequently on "Hardware Hacking" tweets, presentations, and blogposts.

EEPROM is Electronic Erasable Programmable Read Only Memory, which is a mouthful. In short, it's a place we can store configuration data to change the behavior of the Shikra device. Due to the nature of EEPROM the data we store there will persist even after the Shikra has lost power and been unplugged. (We go into quite a bit of detail on PROM, EPROM, EEPROM, and FLASH in SexViaHex so we'll skip it here.) We can do cool things like change the device name, serial number, USB descriptors, and even change the onboard LED to blink on data transmission! The project is open source and available here: https://github.com/Xipiter/shikra-programmer

OTHER NEWS

Read Xipiter Newsletters

RSS Feed

UPCOMING TRAINING

Practical Android Exploitation

Blackhat, Las Vegas 2018
SOLD OUT

2019 - TBA

Software Exploitation Via Hardware Exploitation

Blackhat, Las Vegas 2018
SOLD OUT

2019 - TBA

Practical ARM Exploitation

2019 - TBA

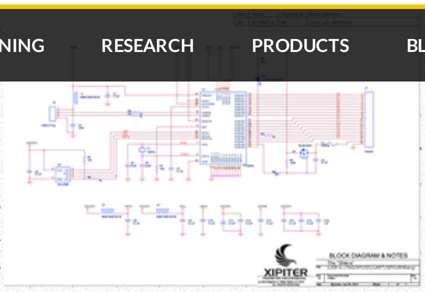
HackAWebcam Workshop

2019 - TBA

BLOG CATEGORIES

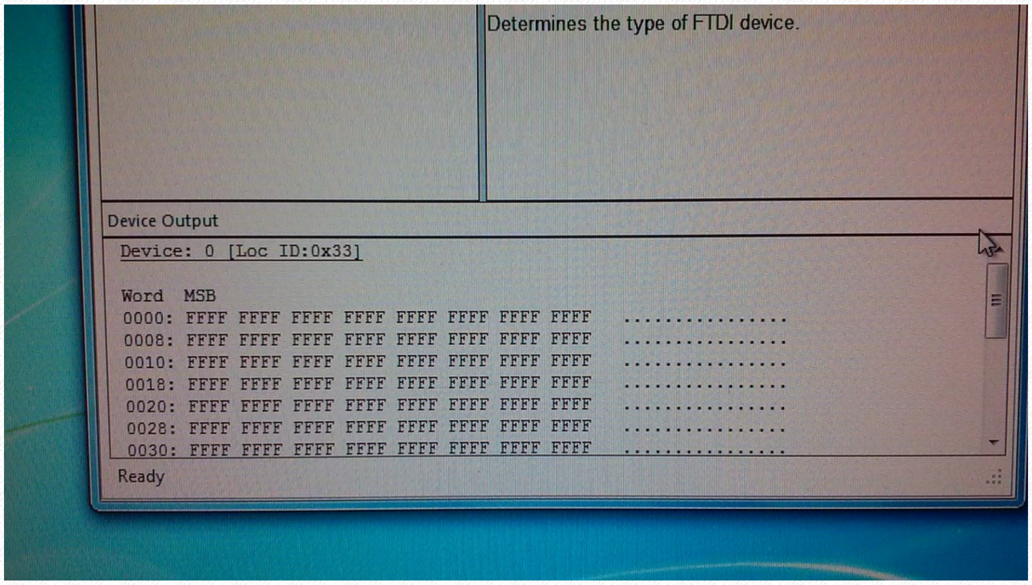
All
Embedded Devices
Exploitation
INT.CC
Iot
Research
The Insecurity Of Things

The Shikra is manufactured with a 93C56 EEPROM holding 128, 16 bit words of data. The EEPROM is a separate chip onboard the Shikra PCB, which is connected to the FTDI FT232H USB controller. We started by pulling data sheets from FTDI, and lots of them. If you have a question, chances are the answer is in a data sheet, somewhere. As you know, the life of a reverse engineer is basically spent in data sheets. The unfortunate part is that these data sheets can often be confusing and hard to read for someone new to the subject. One of the first things we did was get FTDI's FT_PROG utility. This utility can talk to the FT232H and program the EEPROM. We plugged in a Shikra, and sure enough we got a screen full of 0xffff words.



The Shikra block diagram.

ELSEWHERE...



```
Determines the type of FTDI device.

Device Output
Device: 0 [Loc ID:0x33]

Word  MSB
0000: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0008: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0010: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0018: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0020: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0028: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
0030: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF   ................
Ready
```

Cool, this utility program works, but it was hard to use. It also doesn't allow for much instrumentation (although there are some DLLs and COM objects) and there is no source code and the official release only runs on Windows. The website INT3 exists so we can make hardware available and ready-to-use for security researchers, so we embarked on writing a Python programming utility that gave the same functionality as the FT_PROG utility.

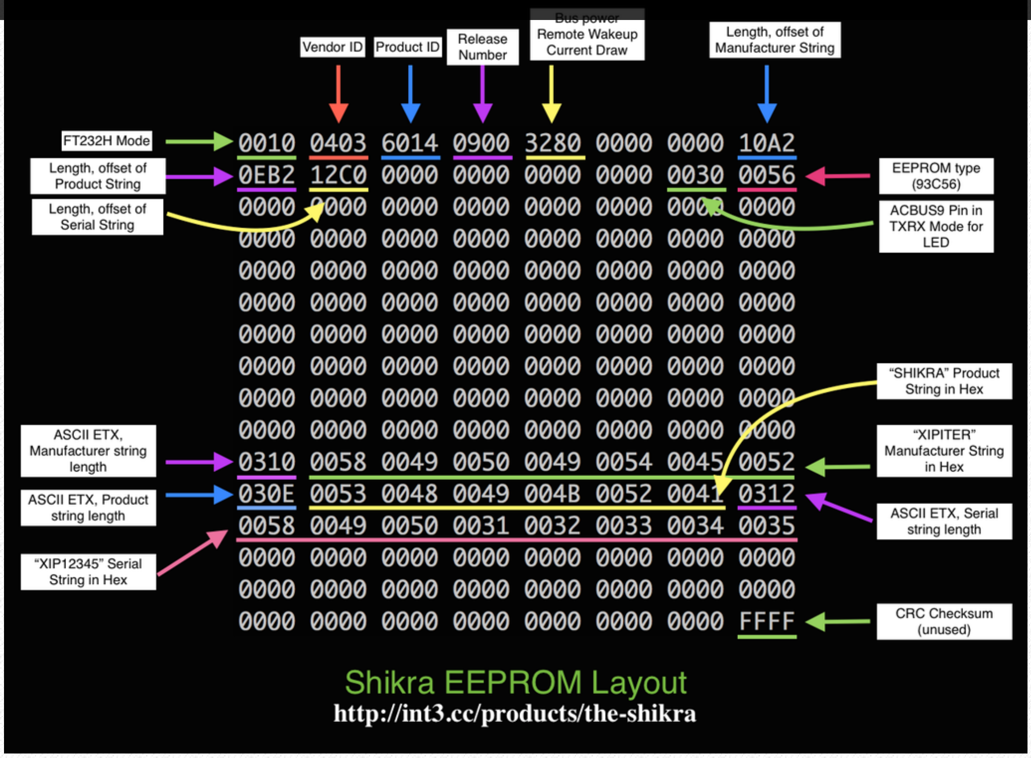
WHERE TO START?

Initially, we just wanted to write *something* to the EEPROM on a Shikra. After that, we could work on writing the right bytes to the right places. We brushed up on USB control transfers and set off. The first thing was to locate the Shikra on the USB bus. The FT232H defaults to Vendor ID 0x403 and Product ID 0x6014. Using pyusb we can search for devices matching those values and see if a Shikra is connected. Once we were able to locate the device, we tried doing USB control transfers to write data to the EEPROM. We were able to write 0x0000 to each word and then verify it worked by reading the EEPROM with FT_PROG. Yay!

EEPROM LAYOUT

I noticed that FT_PROG's hex dump screen was in big endian format, but the bytes when reading and writing were in little endian format. The EEPROM is mentioned in many data sheets, but never the layout of the actual configuration data. Generic statements say that things "are possible" with programming of the EEPROM, but no where is it mentioned *HOW* to do it. Apparently, that's something you have to sign an NDA with FTDI to find out. Good thing a few other people have open source code that documents the EEPROM layout quite well (links are in the header of Shikra programmer code)

Sneak peek of full EEPROM layout:


Vendor ID
Product ID
Release Number
Bus power Remote Wakeup Current Draw
Length, offset of Manufacturer String
FT232H Mode
Length, offset of Product String
Length, offset of Serial String
EEPROM type (93C56)
ACBUS9 Pin in TXRX Mode for LED
"SHIKRA" Product String in Hex
ASCII ETX, Manufacturer string length
"XIPITER" Manufacturer String in Hex
ASCII ETX, Product string length
ASCII ETX, Serial string length
"XIP12345" Serial String in Hex
CRC Checksum (unused)
Shikra EEPROM Layout
http://int3.cc/products/the-shikra

```
0010 0403 6014 0900 3280 0000 0000 10A2
0EB2 12C0 0000 0000 0000 0000 0030 0056
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0310 0058 0049 0050 0049 0054 0045 0052
030E 0053 0048 0049 004B 0052 0041 0312
0058 0049 0050 0031 0032 0033 0034 0035
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 FFFF
```

The full Shikra EEPROM Layout

STARTING PROGRAMMING

My goal was the write a barebones configuration to the EEPROM with my utility, and then read the data back in the FT_PROG utility in a correct way. This way we knew that what we programmed was correct enough for FTDI's standards. Over the next couple of days we spent a lot of time plugging and unplugging the Shikra after tweaking values. We were unable to get the FT_PROG utility running on a Windows VM, so we fired up an old physical desktop to do the FT_PROG tasks (and hence going back and forth between computers. A lot.) The actual EEPROM programming is very boring. "Shikra, write 0x0000 to address 0x0!" And so on and so forth. Reading and writing values was the easy part. It's all about knowing *WHAT* and *WHERE* to write values. We had to come up with a good way to model the Shikra EEPROM memory in Python and then "flush" the data structure to the EEPROM when programming. We ended up using the Struct library and a list. Each list/array element was a single 8-bit word of data. But didn't we say that the EEPROM was addressing in terms of 16-bit words? Yes, we did mention that, which slightly complicates things. Every read or write would grab the two adjacent list elements and consider that one "word" to read or write to EEPROM. We wrote a few utilities that would take two 8-bit bytes and combine them into a 16-bit word and vice-versa.

AN INTERIM ON STRUCT PACKING AND UNPACKING

Everything in the Shikra programming utility uses a linear array of byte strings to represent EEPROM memory. This is literally represented in Python as a list of strings, where each element is a packed Struct object represented as a string. An example would be the 16 "byte" list:

```
['\xff', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x0    0', '\x00', '\x00', '\x00',
```

gistfile1.txt hosted with ❤ by GitHub view raw

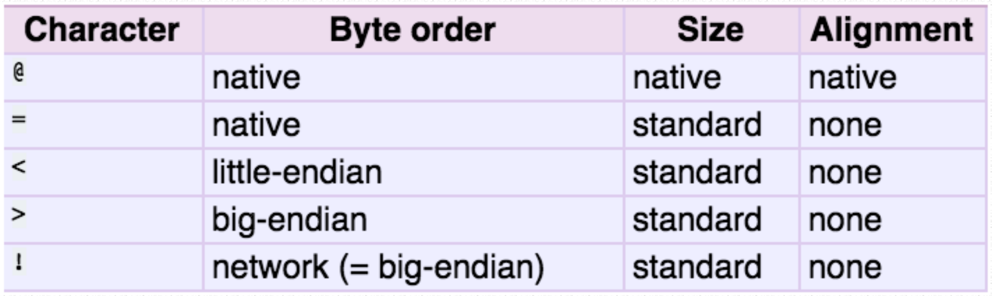
Each element is a struct.packed() value, stored as type string.

```
#! /usr/bin/env python
import struct

SIZE_OF_STRUCT = 16

```

```
temp_eeprom_array = ["\x00" for x in xrange(SIZE_OF_STRUCT)]  # initialize with 0x00 byte strings

index = 0x00
byte_to_write = 0xff
struct_format = '>B'
temp_eeprom_array[index] = struct.pack(struct_format, byte_to_write)

# print temp_eeprom_array
# Output: ['\xff', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00',
# A list of struct.packed hex values, represented as strings.

byte_to_read = struct.unpack(struct_format, temp_eeprom_array[index])

# print byte_to_read
# Output: (255,)  This is a tuple with the hex value being the first index
# This is why in the Shikra programming utility you see a lot of struct.unpack(struct_format, temp_

byte_to_read = byte_to_read[0]

# print(type(byte_to_read))
# Output: type <int>
# This means we can manipulate the hex value, stored in python as an int, and then write it back to
```

gistfile1.txt hosted with ❤ by GitHub view raw

Some further explanation on the '>B' format string for packing and unpacking values:



Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

WRITING STRING CONFIGURATION DATA

Now that we can manipulate, read, and write data, we needed configuration to write to the device EEPROM. The first couple words on the EEPROM are reserved for FT232H mode, Vendor ID, Product ID, Release Number, and Current Draw. The next few words are essentially pointers to data mentioned later in the EEPROM. These pieces of data are the Manufacturer string, Serial Number, and Product String. In the beginning of EEPROM we write the starting address of each string, and it's corresponding length. There is a large gap in EEPROM between these pointers and the actual strings, which we would later find out is where the *real* configuration goes on.

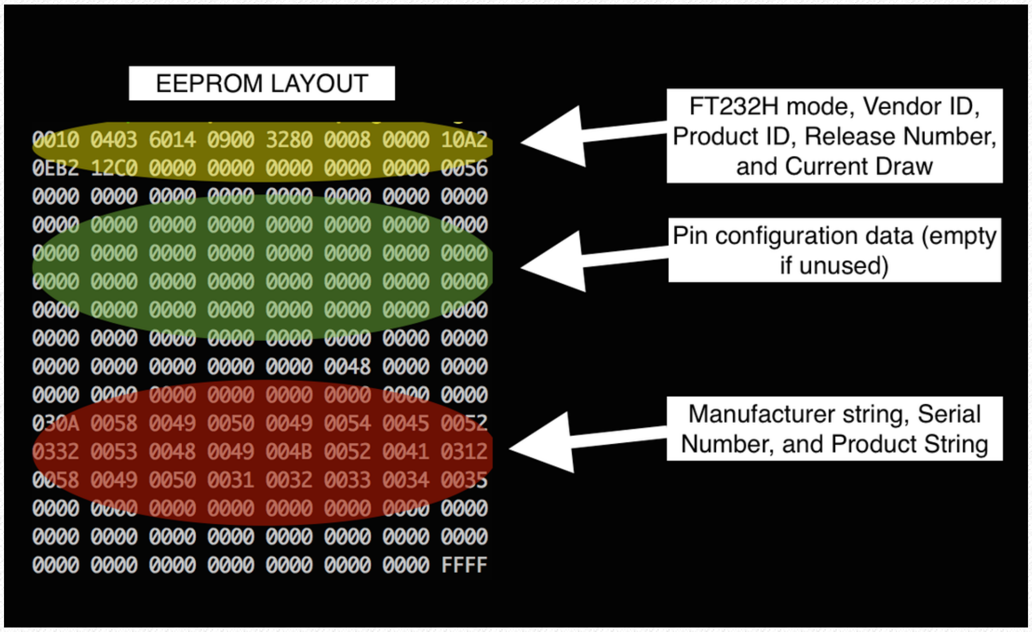
CHANGING PIN OPTIONS

At this point we have a barebones configuration written to the Shikra. We went through a lot of back-and-forth of reverse-engineering the layout specifics through trial and error of FT_PROG. We also looked at as much open source code as possible. Eventually this led to a moment when FT_PROG would read my EEPROM contents perfectly. This was pretty cool, but we hadn't got to the main point — changing board pin configuration through programming. My first goal was to configure the onboard LED to blink.

LED PROGRAMMING

After reviewing the board layout, we determined the LED was connected to ACBUS9 on the FT232H controller (another reason FT_PROG is confusing is that this pin is called C9 within the utility). We programmed the Shikra using FT_PROG, making sure to set the state of the ACBUS9 pin. The pin has 5 states that are applicable to the LED: tx, rx, txrx, tristate, and drive 0. Tx, rx, and txrx blink the led according to data passing by. Tristate is the default 'off' configuration, and drive_0 will turn the LED on consistent. We would set the EEPROM contents with FT_PROG for the LED, and then would dump contents with my programming utility. This way we would know the slight differences between each LED mode, and could record them accordingly.

This pin configuration data is stored in the space between the string pointers and the actual strings.

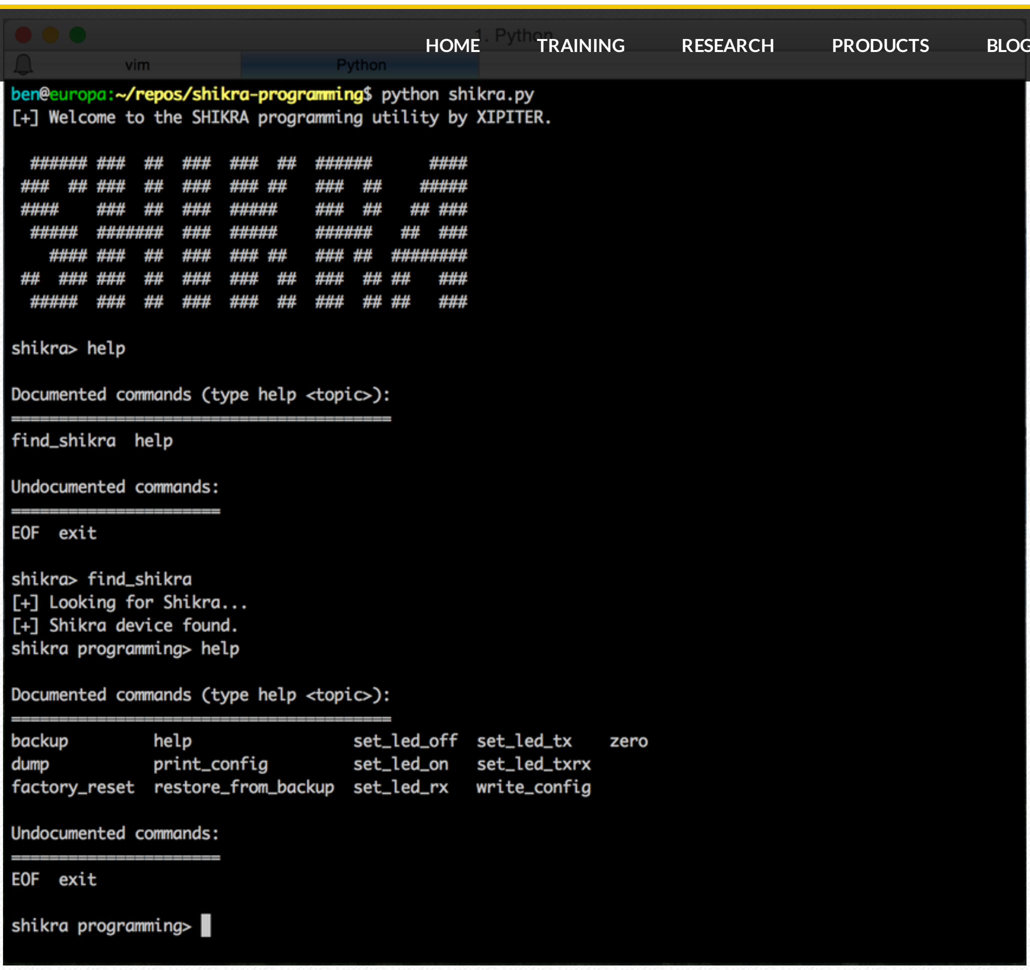

EEPROM LAYOUT
FT232H mode, Vendor ID, Product ID, Release Number, and Current Draw
Pin configuration data (empty if unused)
Manufacturer string, Serial Number, and Product String

```
0010 0403 6014 0900 3280 0008 0000 10A2
0EB2 12C0 0000 0000 0000 0000 0000 0056
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0048 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
030A 0058 0049 0050 0049 0054 0045 0052
0332 0053 0048 0049 004B 0052 0041 0312
0058 0049 0050 0031 0032 0033 0034 0035
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 FFFF
```

Please note that on older Shikra devices, the LED modes may not work!

WRAPPING IT UP ALL NICE

To review, we have been able to write the EEPROM contents with basic configuration data, as well as changing the FT232H pins to different states. we was able to change the operation of the pin connecting to the Shikra's LED to blink under different circumstances. This was nice, but my code was pretty messy and was not very usable. We refactored most of the code, and wrapped it all in a nice Python interactive style interface with the Cmd library (as we do with quite a few Xipiter utilities). This way the utility has built in help, and has straightforward menus items. Here is a screenshot of the utility:



```
ben@europa:~/repos/shikra-programming$ python shikra.py
[+] Welcome to the SHIKRA programming utility by XIPITER.

  ###### ###  ##  ###  ###  ##  ######      ####
 ###  ## ###  ##  ###  ### ##   ###  ##    #####
 ####    ###  ##  ###  #####    ###  ##   ## ###
  #####  ####### ###  #####    ######   ##  ###
    #### ###  ##  ###  ### ##   ### ##  ########
 ##  ### ###  ##  ###  ###  ##  ###  ## ##   ###
  #####  ###  ##  ###  ###  ##  ###  ## ##   ###

shikra> help

Documented commands (type help <topic>):
========================================
find_shikra  help

Undocumented commands:
======================
EOF  exit

shikra> find_shikra
[+] Looking for Shikra...
[+] Shikra device found.
shikra programming> help

Documented commands (type help <topic>):
========================================
backup         help                set_led_off  set_led_tx    zero
dump           print_config        set_led_on   set_led_txrx
factory_reset  restore_from_backup set_led_rx   write_config

Undocumented commands:
======================
EOF  exit

shikra programming> █
```

I hope the Shikra programming utility can be of use to you. We appreciate feedback and comments. This project helped me brush up on hexadecimal format, bit-shifting, the USB protocol, and many other things along the way. We got a good amount of hardware review and experience debugging electrical circuits with my multimeter. There were ups and downs, but overall the outcome is pretty slick!

ABOUT THE AUTHOR

Ben Reichert (@tgjamin) is formerly a Security Engineer at Xipiter. He is currently entry-level security Engineer with Senrio Inc. His background is in System Administration, DevOps, and Mobile security. Previously he has worked in large automation environments and maintained massive-scale websites. He has found a number of vulnerabilities in embedded systems, and Android applications. Ben enjoys reverse engineering, mobile web security, penetration testing, and hardware hacking. He (along with all of the Xipiter staff as of Jan 2016) is currently working on Senrio: enterprise-grade security for embedded systems.

Comments are closed.