Fuzzing USB devices using frisbeeLite

Andy Davis
Research Director
andy.davis 'at' ngssecure.com

An NGS Secure Research Publication

13 January 2012

http://www.ngssecure.com

Table of Contents

1. Introduction

At Black Hat USA 2011 I presented "USB – Undermining Security Barriers"[1], which detailed a fuzzing approach that enabled USB devices and hosts to be security tested in a platform-independent manner. However, this approach required the use of USB test equipment hardware in conjunction with bespoke fuzzing software (Frisbee). Since then I have needed to fuzz more USB devices than USB hosts and therefore decided to develop simple fuzzer that could be used to test them.

Frisbee Lite has been written in wxPython for the Windows platform, although only relatively minor changes would be required to port it to Unix-based platforms. It is a "dumb" fuzzer in that it requires the user to understand the types of USB request packets that are likely to trigger security flaws, but just running it with minimal knowledge of the USB protocols would have discovered the two USB bugs that were used to jailbreak various Apple products in recent years.

This paper will discuss the format of device requests that are sent to USB devices in order to hopefully provide an insight into areas where software flaws may exist. It will also discuss a number of public vulnerabilities in USB devices and finally, the installation and usage of Frisbee Lite.

2. Communication with USB devices

This section contains an overview of how communication is performed with a USB device. Much of the information presented here is also available in the USB Specification v2.0[2]. All USB devices respond to requests on the device's Default Control Pipe. The requests are made using control transfers and the parameters are sent to the device in a Setup packet. Table 1 shows the format of a setup packet.

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Value	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Table 1: USB Setup packet format

2.1. bmRequestType

This is a bitmapped field that describes the characteristics of the request. For example, it identifies the direction of the data transfer in the second (data) phase of the control transfer (the direction bit is ignored if the wLength field is set to zero, hence implying that there is no data stage). There are a number of standard requests (see Table 2) within the USB specification, in addition to class-specific requests. Requests can be sent to a USB device, an interface or an endpoint (on the device). Therefore, the *bmRequestType* field also includes information about the intended recipient. If the recipient is an interface or endpoint the *wIndex* field specifies the interface or endpoint.

2.2. bRequest

This is the actual request that is being sent (the "Type" bits in the *bmRequestType* field change the meaning of this field). Standard requests are detailed in Table 3.

2.3. wValue

The contents of this field are request-specific.

2.4. wIndex

The contents of this field are request-specific. However, it is often used to specify an endpoint or interface.

2.5. wLength

This specifies the length of the data transferred during the second phase of the control transfer. If this field is zero, there is no second (data transfer) phase. On an input request, a device should never return more data than is indicated by the *wLength* value; it may return less. On an output request, *wLength* should always indicate the exact amount of data to be sent by the host.

2.6. Standard device requests

There are a number of standard device requests, which are the same for all USB devices; these are detailed in Table 2. USB devices must respond to standard device requests, even if the device has not yet been assigned an address or has not been configured.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Table 2: Standard requests

The values associated with the standard request codes, e.g. *GET_DESCRIPTOR*, used in Table 2 are shown in Table 3.

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	*2*
SET_FEATURE	*3*
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Table 3: Standard request codes

The values associated with the USB descriptor types used in Table 2 are shown in Table 4.

Descriptor	Value
DEVICE	1
CONFIGURATION	2
STRING	*3*
INTERFACE	*4*
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Table 4: Descriptor types

The values associated with the standard feature selectors used in Table 2 are shown in Table 5.

Feature selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	Device	1
ENDPOINT_HALT	Endpoint	0
TEST_MODE	Device	*2*

Table 5: Standard feature selectors

As can be seen, the number of different permutations of device request is huge and in a number of cases, unusual combinations of values supplied in these requests have led to situations where the device request parsers in USB device drivers have not been capable of processing them, resulting in an exception or kernel panic. This has led to various publicly disclosed security vulnerabilities in USB devices.

3. Public USB device vulnerabilities

A number of USB device vulnerabilities have been publicly disclosed and some have been subsequently exploited. The two most high profile vulnerabilities[∞] relate to Apple products and are known as:

- usb_control_msg(0xA1, 1) or "steaks4uce" exploit
- usb_control_msg(0x21, 2) exploit

As can be seen from the titles, the device request is being sent with the *bmRequestType* set to the first value and *bRequest* set to the second value. More information about these vulnerabilities can be found on the iPhone Wiki[3].

3.1. usb_control_msg(0xA1, 1)

A heap overflow exists in the iPod touch 2G boot ROM's DFU (Device Firmware Upgrade) mode when sending a USB control message of *bmRequestType* = 0xA1, *bRequest* = 0x1. On newer devices, the same

[∞] *NGS Secure played no part in either the discovery or exploitation of these vulnerabilities.*

USB message triggers a double free() vulnerability when the image upload is marked as finished, also rebooting the device (but this second vulnerability is not exploitable).

3.2. usb_control_msg(0x21, 2)

A null pointer dereference vulnerability exists in the versions of iBoot/iBSS/iBEC found in firmware versions 3.1/3.1.1 and 3.1.2 on all iDevices. The vulnerability existed because of a missing check of the contents of a processor register. Often null pointer dereference vulnerabilities cannot be exploited, however in this instance it can because the MMU (Memory Management Unit) maps whatever is running (LLB, iBoot, etc.) to address zero so that if an exception vector is triggered, it would jump to the one designed to be used with what is running, as opposed to jumping to what is normally located at address zero, the boot ROM.

So, it can be seen that exploitable vulnerabilities can be discovered in the driver software running on USB devices, which led to the development of Frisbee Lite.

4. Frisbee Lite

Frisbee Lite is a "dumb" USB device fuzzer – i.e. the intelligence is in the user. It enables any single USB device request to be created and sent or multiple requests iterated through using a brute-force fuzzing approach. Based on the information presented in Section 2, the reader should now have a clearer idea of values to set within Frisbee Lite in order to create situations where a software flaw (and potential security vulnerability) may lie.

4.1. Software download

There are a number of prerequisites that need to be downloaded and installed in order to use Frisbee Lite. These are detailed in this section.

- Download and install Python (if you haven't already got it) - http://www.python.org/getit/
- Download and install wxPython - http://www.wxpython.org/download.php#stable
- Download and extract FribseeLite.zip - http://www.ngssecure.com/research/research-overview/Public-Tools.aspx

4.2. Installation

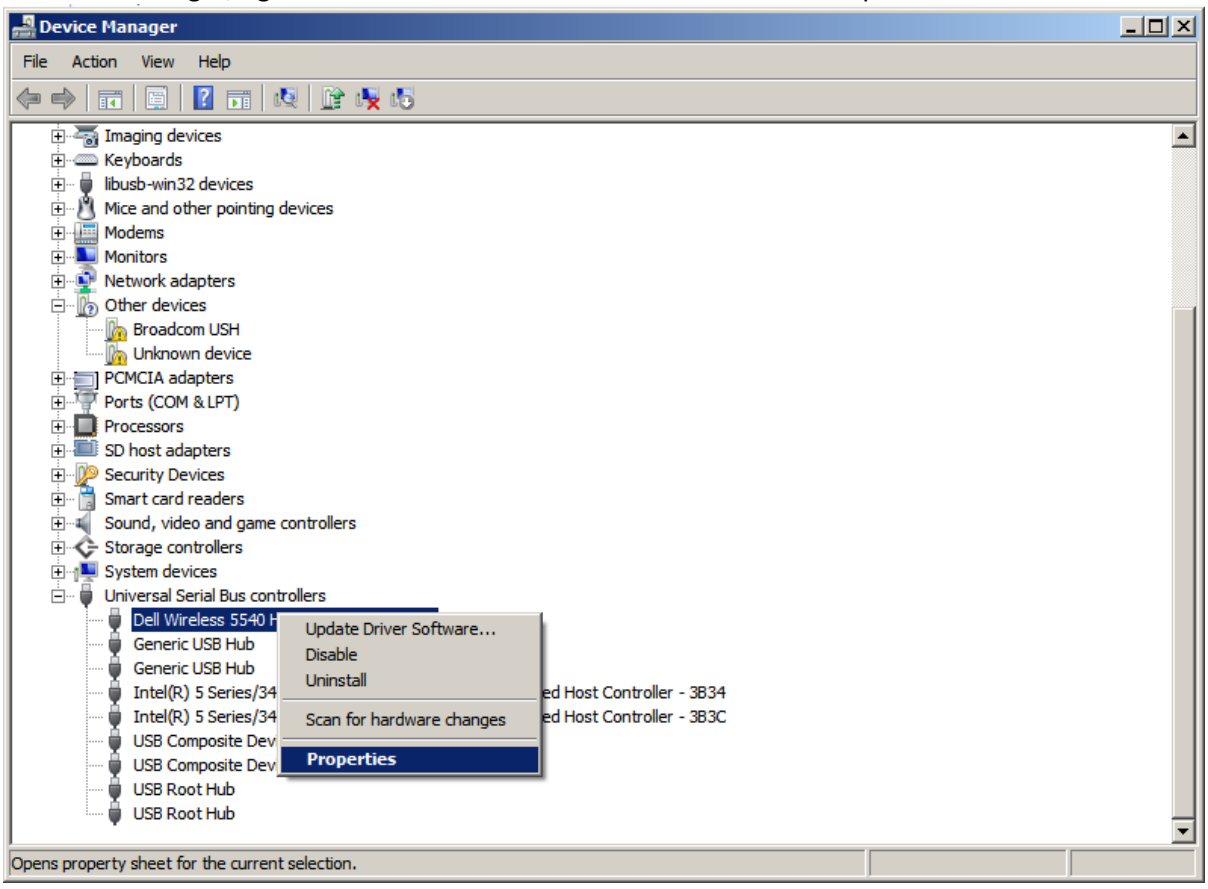
After extracting the Frisbee Lite zip file, from the “dependencies” directory, install *pyusb*:

Extract the zip and type:

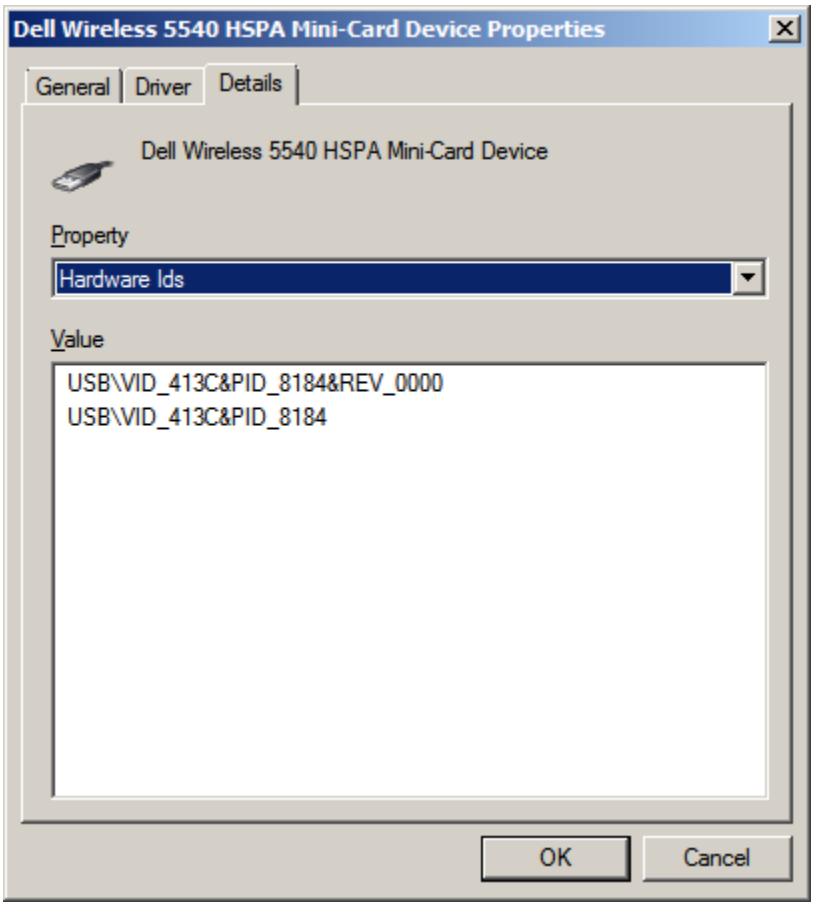
```
python setup.py install
```

Next, the PID (Product ID) and VID (Vendor ID) of the device to be fuzzed must be identified - these are the unique values that identify the device to your PC.

In Device Manager, right click on the device to be fuzzed and select “Properties”:

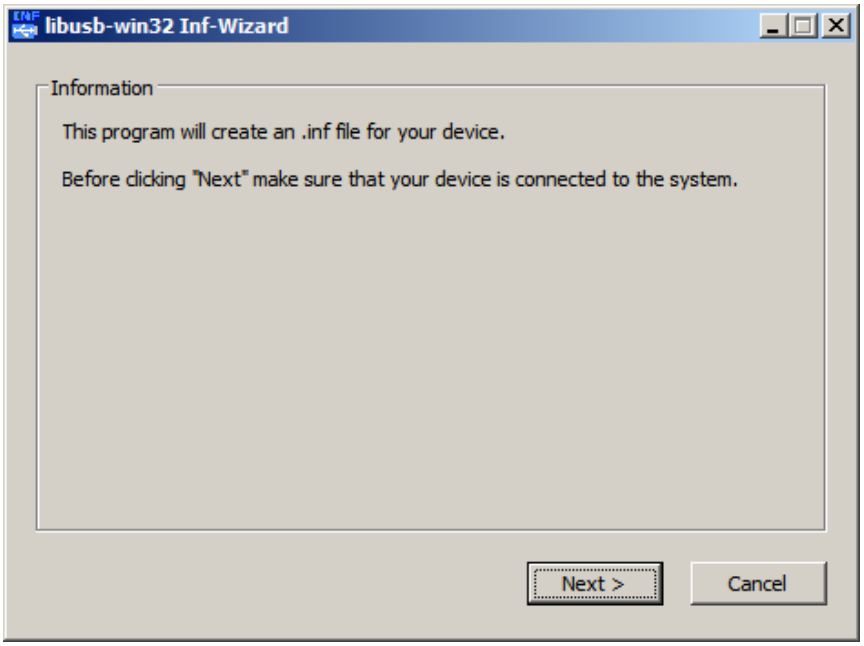


Then select the “Details” tab and select “Hardware Ids”:

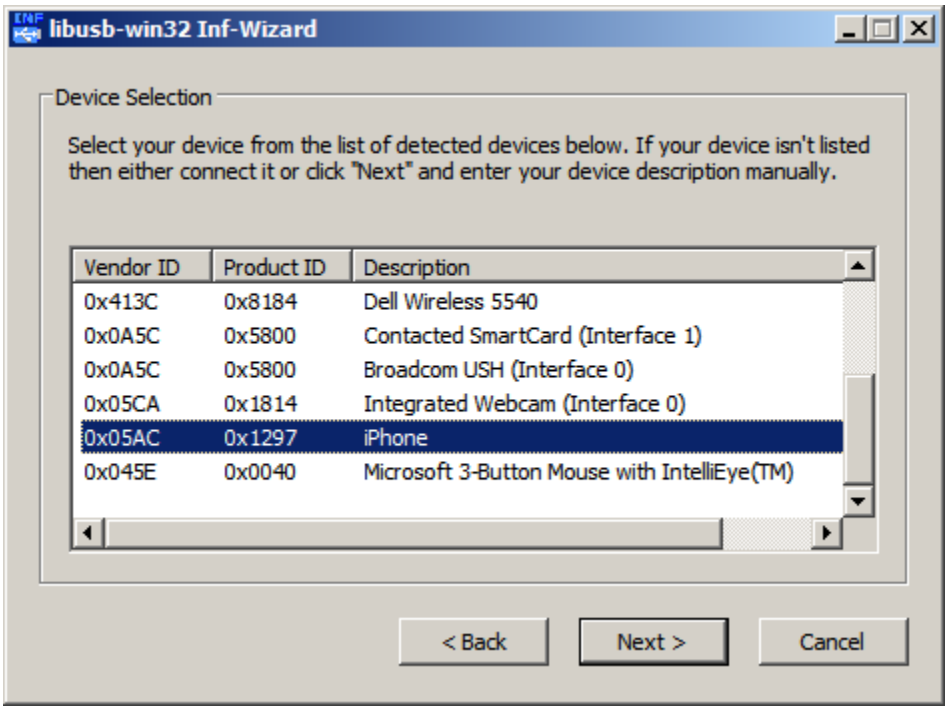


In the example above, the PID = 0x8184 and the VID =0x413c

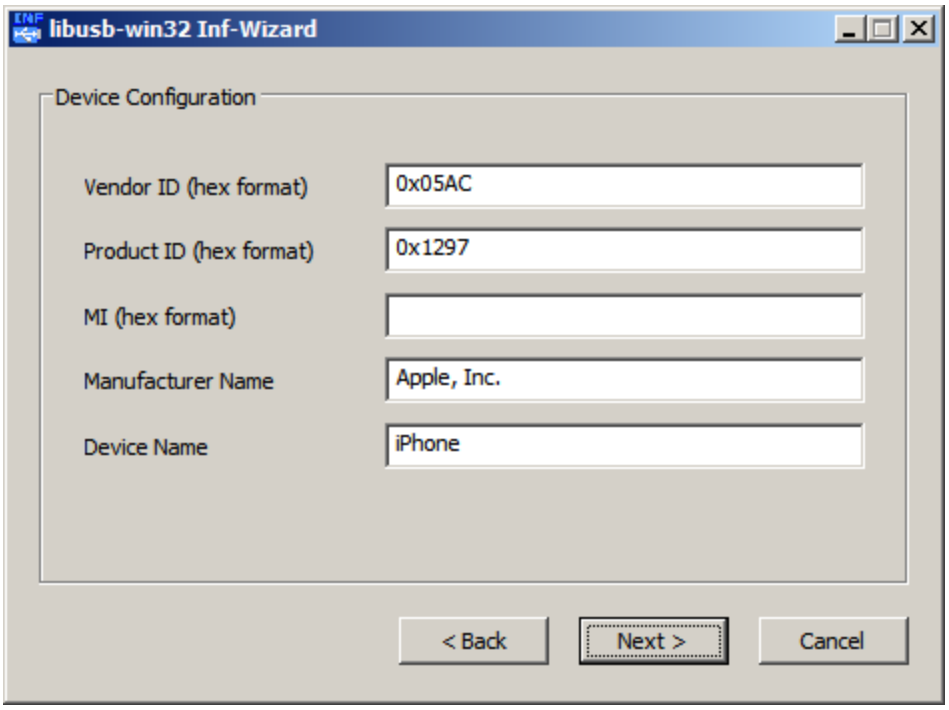
From the “dependencies” directory, extract *libusb* and in the “bin” directory run “inf-wizard.exe”:



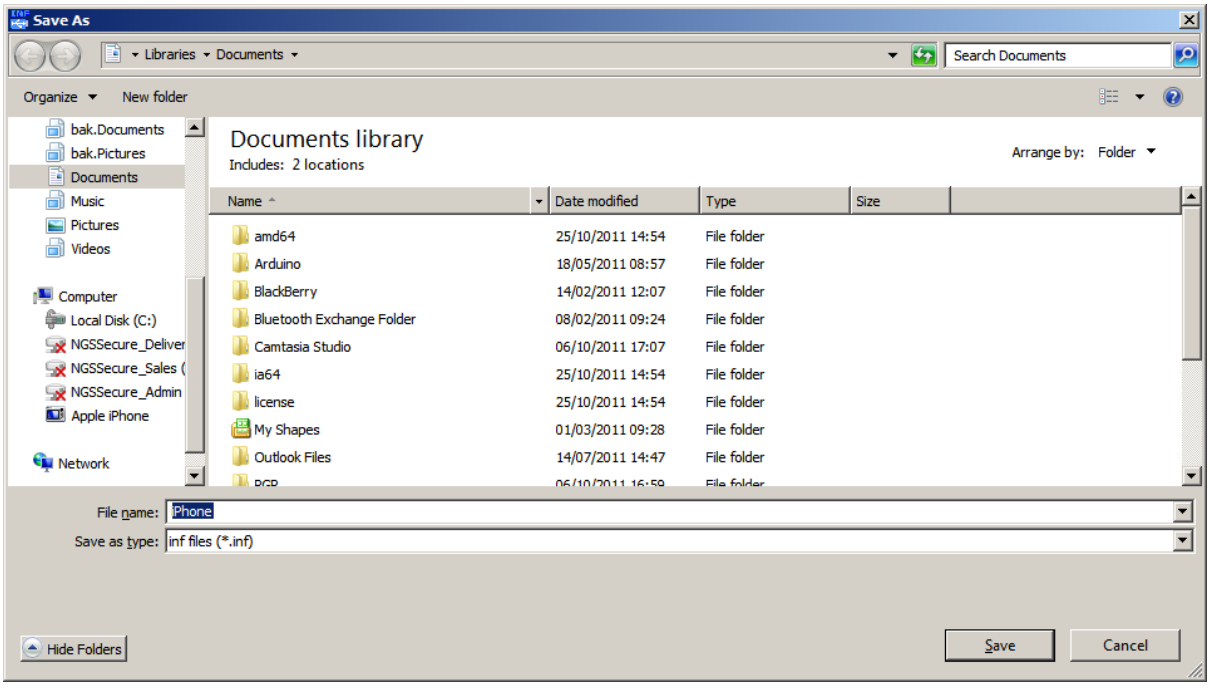
Click “Next”



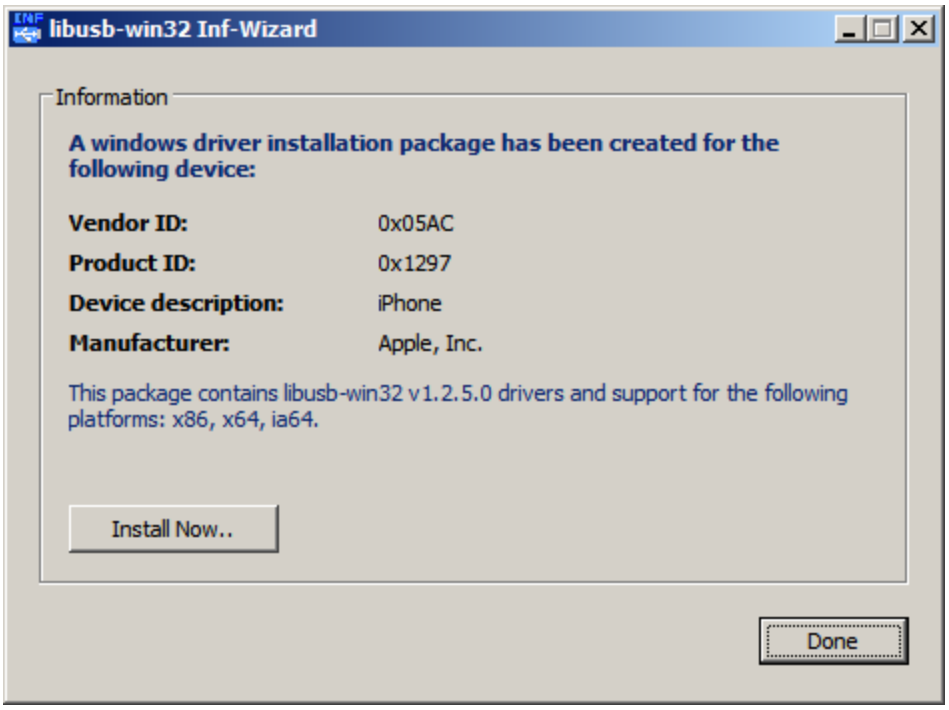
Select the device to fuzz and click “Next”



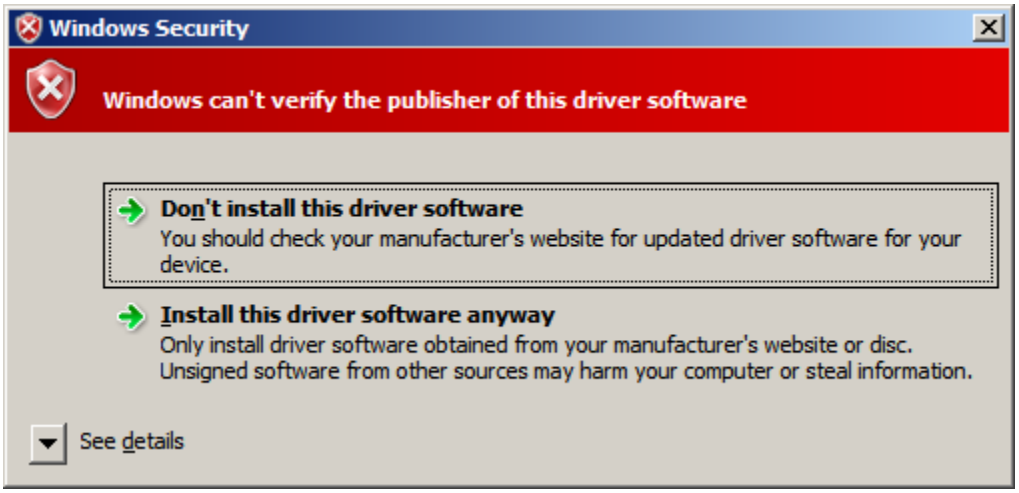
Verify that the PID and VID identified earlier are correct for the device to be fuzzed. Click “Next”



Click “Save”



Click “Install Now”

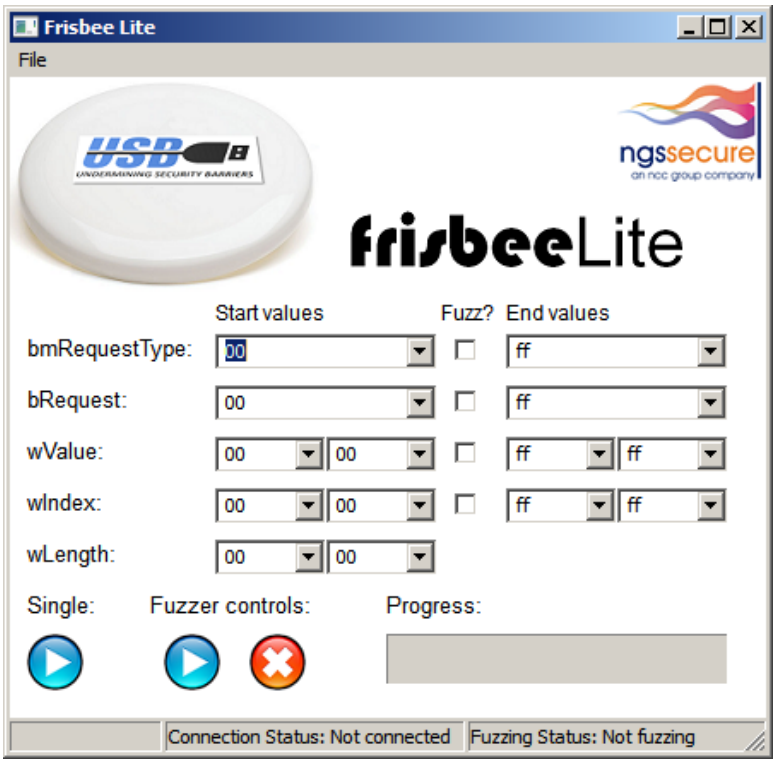


Click “Install this driver software anyway”

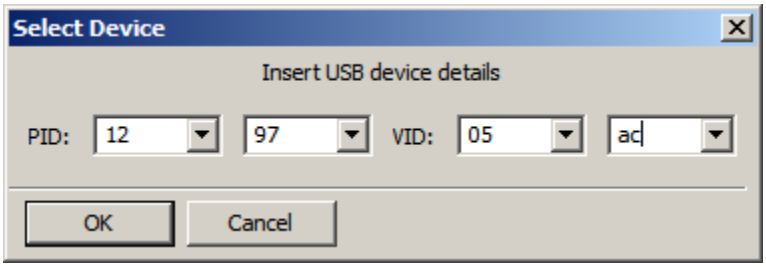
Everything should now be installed.

4.3. Usage

Run “FrisbeeLite.py” and the GUI below should be displayed:

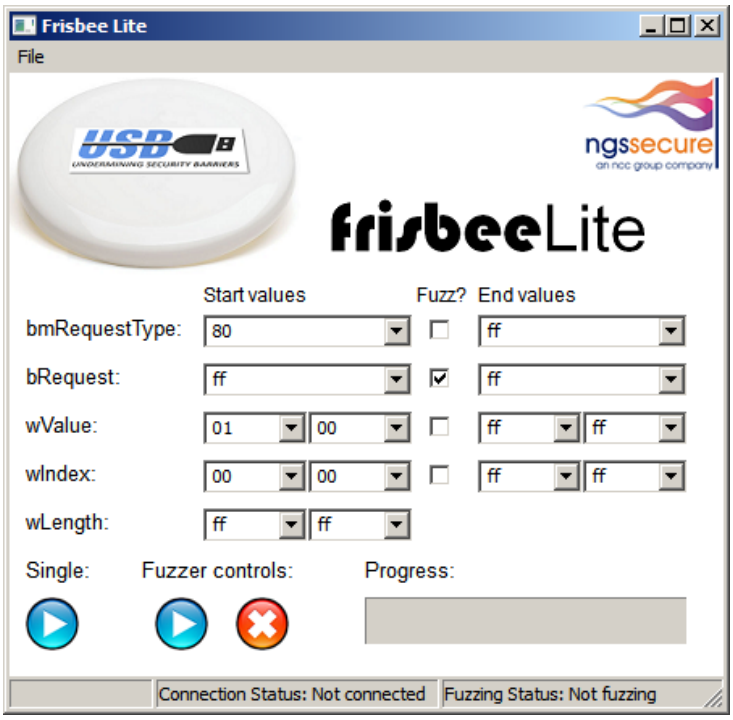


The first step is to select the USB device that will be fuzzed. Click “File” -> “Select USB device”:

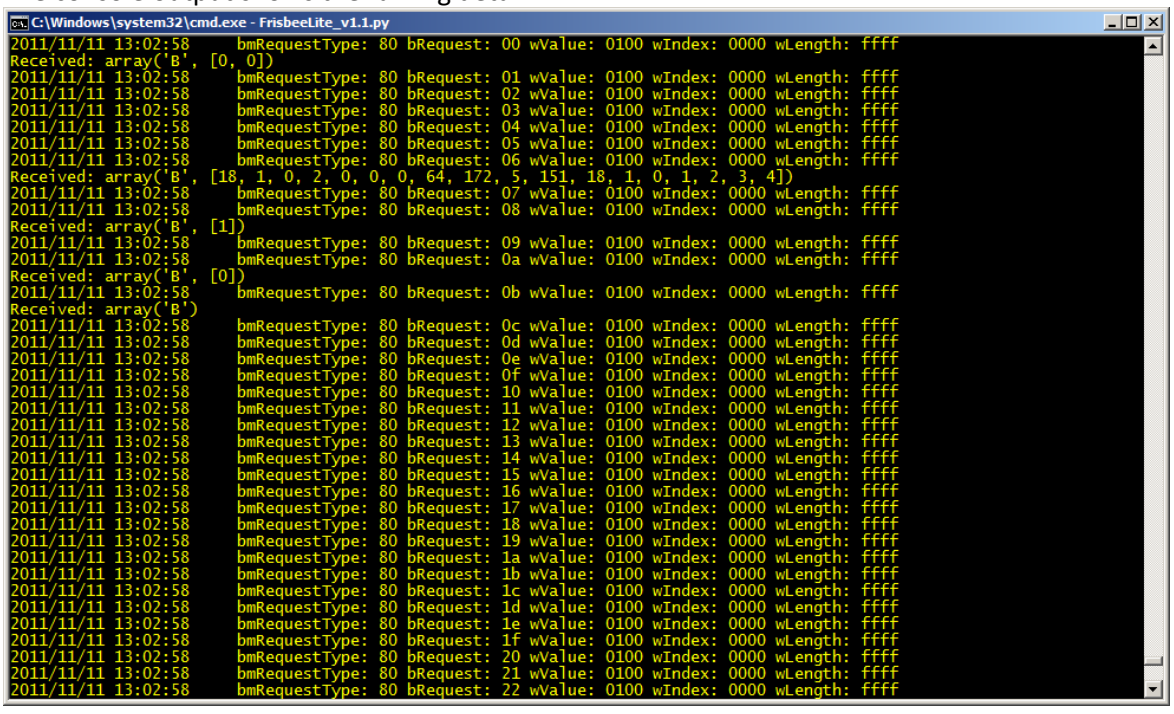


Enter the PID and VID values for the device and click “OK”

All elements within a USB device request can be fuzzed, although it was considered that fuzzing through all the *wLength* values would most likely prove fruitless and therefore, a static value can be set for this field. Fuzzing operation is simple, the values which are to be fuzzed are selected using the checkboxes, the start and stop values are then chosen and the start button is pressed e.g.



The console output shows the fuzzing detail:



The output is also written to a log file in the current directory.

Finally, the “Single” button allows a single USB request to be sent using the currently selected values.

5. Conclusions and further research

There are a large number of different USB device requests that can be generated and sent to a device under test, which is why fuzzing is an appropriate approach to security testing USB devices. Although no inherent "intelligence" has been designed into Frisbee Lite, it still provides powerful capabilities to identify software flaws and potential security vulnerabilities.

Possible additions for future versions of Frisbee Lite may include:

- Instrumentation using either ICMP to check if the device is still accessible over the network or first establishing a "known good" request that results in a repeatable response in order to check if the USB stack is still functioning on the target device
- A greater degree of granularity in the way that the *bmRequestType* values are fuzzed, to make the tool more intuitive to use
- The inclusion of specific test cases that are known to be likely to trigger software flaws in USB driver software.

Hopefully, the tool will be useful to security researchers and pentesters. Any feedback can be provided to me directly via the email address at the beginning of the paper.

6. References and further reading

1 - http://www.ngssecure.com/Libraries/Document_Downloads/USB_-_Undermining_Security_Barriers-BlackHat-USA-2011-Andy_Davis-NGS_Secure.sflb.ashx

2 - http://www.usb.org/developers/docs/usb_20_101111.zip

3 - http://theiphonewiki.com/wiki/index.php?title=Main_Page