Mouse Trap: Exploiting Firmware Updates in USB Peripherals

Jacob Maskiewicz*, Benjamin Ellis, James Mouradian, Hovav Shacham†
UC San Diego

Abstract

Although many users are aware of the threats that malware pose, users are unaware that malware can infect peripheral devices. Many embedded devices support firmware update capabilities, yet they do not authenticate such updates; this allows adversaries to infect peripherals with malicious firmware. We present a case study of the Logitech G600 mouse, demonstrating attacks on networked systems which are also feasible against air-gapped systems.

If the target machine is air-gapped, we show that the Logitech G600 has enough space available to host an entire malware package inside its firmware. We also wrote a file transfer utility that transfers the malware from the mouse to the target machine. If the target is networked, the mouse can be used as a persistent threat that updates and reinstalls malware as desired.

To mitigate these attacks, we implemented signature verification code which is essential to preventing malicious firmware from being installed on the mouse. We demonstrate that it is reasonable to include such signature verification code in the bootloader of the mouse.

1 Introduction

In July of 2013, the US Commerce Department's Inspector General released a report revealing that the US Economic Development Administration (EDA) spent $2.7 million in December 2011 in response to a malware infection [34]. The EDA had intended to destroy all of its IT equipment — every computer, mouse, keyboard, printer, camera, and monitor in their possession, valued at over $3 million — and stopped only when they exhausted their budget, with $170,500 of equipment actually destroyed. The precautions taken were due to a misunderstanding across internal departments, leading administrators to believe they were under a large-scale and sophisticated cyber attack; in reality, only a handful of computers were compromised, and the malware appeared to be untargeted and easily quarantinable. Various technology news organizations ridiculed the EDA for its lack of cybersecurity understanding, believing that peripherals would not have been compromised during the incident [23, 13]. The Verge, a technology magazine, observed that "throwing away computer mice seems like a poor approach to ridding an organization of digital threats" [23]. Ars Technica, a publication generally respected for its technical accuracy, published an article titled "US agency baffled by modern technology, destroys mice to get rid of viruses" [10].

These comments demonstrate the common belief that peripherals such as mice cannot be infected by malware. In this paper, we show that this belief is false. We give, for the first time, an end-to-end demonstration of mouse-borne malware. We show how unprivileged software running on a PC host can replace the firmware on an off-the-shelf Logitech G600 gaming mouse, and how compromised firmware on the mouse can in turn reinfect the PC host.

Far from being benign, mice make an ideal vector for compromise of hardened or air-gapped computers. A compromised mouse could be carried by an unwitting user into a secure facility. Mouse-to-host attack, like keyboard-to-host attack [12], is simple and compact due to the capabilities of the USB human interface device (HID) stack.

We believe that mice, like other embedded devices, should verify the integrity of firmware updates cryptographically. We show that it is possible to implement RSA signature verification within the space constraints of a G600 mouse.

1.1 Background and Motivation

As antivirus software becomes more sophisticated, persistent delivery of malware becomes increasingly difficult. Users are unlikely to suspect peripherals such as keyboards and mice to be capable of containing malware, and modern anti-virus software is often incapable of scanning peripherals' firmware. With both users and defensive software unaware of its presence, malware located in a peripheral could have a long and undetected life.

Malware positioned within a USB peripheral can input HID codes to the computer on behalf of the user. Many modern operating system protections and isola-

*jmaskiew@ucsd.edu
†{bellis,jmouradi,hovav}@cs.ucsd.edu

tions can be circumvented in this manner since such inputs are considered user-intended behavior. Additionally, the attacker-controlled peripheral can retrieve malware from the web and install it on the host machine, allowing attackers to persistently reinstall removed malware and update out-of-date malware. Furthermore, peripherals used across multiple computers can first be compromised on an insecure machine and then be used to compromise a secure machine.

1.2 Related Work

As antivirus software develops, people have put a great deal of effort into delivering malware to locations undetectable by traditional antivirus scanning techniques. Advanced rootkits such as Mebroot are executed before the user's operating system even boots, bypassing many malware detection and prevention mechanisms [33].

USB HID as a delivery mechanism. The USB HID class has a standardized protocol for communication with keyboards, mice, and an array of other USB devices [32]. Several projects have developed custom HID hardware for exploit delivery. In 2010, a briefing on how to write software complying to the USB HID specification was given at BlackHat, demonstrating how to construct a USB device that uses the USB HID protocol to emulate a legitimate user [30]. Crenshaw demonstrated the feasibility of a malicious USB HID dongle and showed how to lock Windows systems down against the installation of new USB devices [14]. There are a number of programmable HID USB keystroke dongle (PHUKD) designs and tutorials available on the web. PHUKD allows an attacker to deliver keystrokes to a user's computer by inserting the dongle into an available USB slot, ultimately gaining control of the computer [28]. A radio frequency version of the device, URFUKED, has been shown to allow adaptive, remote delivery of HID keystrokes to a computer [17].

Attacks on embedded device firmware. Firmware update attacks, or the process of compromising a victim's embedded device by installing firmware of attacker construction, have been demonstrated in several arenas. Cui et al. reverse-engineered HP laserjet printer firmware, leveraging its insecure update mechanism to allow the delivery of firmware to the printer of their choosing [15].

Additionally, Cui et al. provided a survey indicating that insecure update mechanisms were not restricted to the printers used to demonstrate their attack [15]. Even more generally, Belissimo et al. surveyed the security of software updates and found secure software updates to be challenging to implement and make available in practice [6].

Such attacks can apply to embedded systems that are part of traditional PCs. For example, Miller showed how to modify the firmware on Apple laptop batteries [26]. In 2009, Chen successfully reverse-engineered and exploited a firmware updater for a full-size, wired Apple keyboard, allowing delivery of a persistent rootkit to the keyboard itself [12]; as Chen observed, the keyboard could then compromise its host through USB HID. Brocker and Checkoway demonstrated a sophisticated attack to enable a Macbook camera's recording mode without illuminating the LED [11]. The reprogrammed camera could also masquerade as a USB HID device, enabling VM escape, though the camera firmware does not persist across reboots.

Our findings are analogous to Chen's, but for an off-the-shelf mouse rather than keyboard. Compared to Chen, we provide an end-to-end demonstration (it is not clear that Chen implemented his proposed keyboard-to-host attack); in addition, we show that cryptographic protection against unauthorized firmware is feasible on the mouse.

Malicious mice. There has been a fair amount of work outside of academia involving mice specifically; these attacks often rely on modifying the hardware of the mouse.

Several individuals have used the pixel array presented by the optical laser on the mouse to turn it into various kinds of low-definition cameras. Franci Kapel turned a mouse into a web camera while Jeroen "Sprite_tm" Domburg combined the pixel array with movement information to turn it into a scanner [22, 16].

In March 2014, German technology magazine *c't* reported a novel trigger for sending the payload. The mouse waits to deliver the payload until the 18×18 pixel matrix read in by the optical laser matches a particular pattern [3, 8].[1]

This inspired Imgur user Indyaner to experiment with adding custom hardware to mice in order to deliver malware. Indyaner added custom hardware to a Logitech mouse and repackaged it [19, 7]. Notably, the Arduino that Indyaner used has an ATmega32u4 which is very similar to the ATmega32u2 that we found inside of the Logitech G600 gaming mouse.

Netragard conducted a devastating pentest involving modified mice. After modifying the mouse's hardware, they repackaged it and mailed it to a target in the company with fliers to disguise the mouse as a promotional gift. The target employee proceeded to plug the mouse in at work, causing the mouse to contact Netragard's command and control server [29]. This validates our statement that a mouse is a valid vector for infiltrating an organization.

1.3 Attack Model

There are two components to our attack: infecting the mouse and exploiting the target. In this section, we present several options for both of these components.

1.3.1 Infecting the Mouse

Ensuring that an infected mouse is attached to the target machine requires different steps depending on whether the target is networked or air-gapped.

The networked case is simpler since the mouse is already connected to the target machine. The mouse firmware must be updateable, and the adversary must control a program capable of updating the mouse running on the target's machine. Since USB is not a protected resource, the adversary's program only needs user-level privileges. Common techniques for obtaining such user-level privileges include web exploits, drive-by-downloads, and distributing Trojan horses in repackaged software. Alternatively, an attacker could deliver users a seemingly legitimate mouse firmware update via a man-in-the-middle attack, posing as the mouse manufacturing company. We note that the original firmware updater we modified was delivered over HTTP, which would enable this kind of attack [25].

The second attack method is to deliver an already-compromised mouse, prepared in a lab environment, to a victim computer. A known infiltration technique is to leave compromised USB peripherals, such as USB mass storage devices, in a company parking lot [31]. Users find the storage devices and use them, assuming they were incidentally dropped by a benign individual. A compromised mouse found in a shopping bag, repackaged, with the receipt included, could just as easily be taken by an unsuspecting user, as demonstrated by the Netragard pentest [29]. A common defense against USB mass storage device attacks is to disable the Windows AutoRun.exe service; this mitigation is ineffective against mice.

1.3.2 Delivering the Malware

We consider two separate attack models: one where the mouse is connected to a machine with Internet access and another where the mouse is connected to an air-gapped machine.

If the mouse is connected to a machine with Internet access, then the mouse can simply open a shell in order to download and run malware. This also allows the mouse to retrieve updated versions of the malware or reinstall malware if it has been detected and removed. This attack model involves only low-complexity firmware modification on the mouse.

If the mouse is connected to a machine without Internet access, then the mouse must contain the entire malware. It can then transfer this malware to the host machine over USB. Some possibilities for this transfer mechanism include: 1) constructing the malware using shell utilities directed by key commands (see §5 for more details), 2) identifying as a firewire-to-usb adapter and using direct memory access [5], and 3) reporting as a usb storage device with the executable onboard and clicking on it . This attack model requires that the mouse is either shared between secure and insecure computers or that the mouse is attached to a secure computer after being delivered surreptitiously as described above.

Furthermore, peripherals are often reused when new computers are acquired. A consumer may buy a new desktop but reuse their old keyboard, monitor, mouse, and speakers. In this way an infected mouse may transfer across generations of computers.

1.4 Roadmap

To mount our attack, we first obtained a mouse whose firmware can be updated and therefore compromised. We describe this mouse and its architecture in §2. We then reverse-engineered the mouse firmware with standard techniques as we describe in §3. We proceeded to reverse-engineer the firmware updater and patch it to enable flashing the mouse with firmware of our construction, as we will discuss in §4. We describe our various current exploits in §5; they all revolve around sending HID codes to a terminal to execute arbitrary commands and infect the target computer with malware. In §6 we discuss potential alternate triggering mechanisms for our attack. In §7, we propose mitigations to our attack and discuss their feasibility.

1.5 Ethics and Disclosure

We notified the vendor of our findings on July 6, 2014. We do not believe that the problems we identified are unique to the mouse we studied. Rather, we believe that every device with flashable firmware is potentially at risk. Mice and keyboards are assumed to speak for the user, simplifying device-to-host attacks; but other devices, once compromised, can reinvent themselves as keyboards [11].

2 Overview of the Mouse

In this section we will describe the architecture and regular behavior of our chosen mouse.

2.1 Choice of Mouse

For both ease of programming and wider spread compromise capability, we elected to use a mouse that has a software firmware update mechanism. After finding updateable mice via the web, we examined two such mice: the Logitech G500s and G600. Manual disassembly and inspection of the microcontrollers revealed that the Logitech G600 features an Atmel ATmega32u2 which has a readily available data sheet and runs AVR, an 8 bit RISC architecture. The Logitech G500s, however, appeared to have a custom microcontroller and at the time of this report did not appear to have publicly available programming information. Therefore, we chose the G600 as our target.

2.2 Mouse Firmware Structure

The ATmega32u2 uses a Harvard architecture, so the chip's memory is split into program and data memory. Within the application section, the firmware is split into two segments: an application segment and a bootloader segment. The application segment starts at the beginning of program memory whereas the bootloader segment occupies the last $1/8$ of program memory.

The application segment contains all of the code to be run during normal operation of the mouse. This includes the mouse's main sleep loop, SPI interrupt handling from the optical laser, and sending both USB HID codes and button presses to the host.

The bootloader is responsible for managing the firmware update process. As part of this responsibility, it has privileges that the application segment does not necessarily have. For instance, the bootloader is able to write to application memory [1]. The firmware update process does not write to the bootloader segment of the mouse, so it is always possible to flash the mouse with firmware. This proved valuable since it means that a buggy firmware update does not permanently disable the mouse.

2.3 Sending USB HID Codes

Because the G600 is a gaming mouse, it has far more buttons than the standard left button, right button, and scroll wheel. It has an entire number pad on its side (labeled G9 through G20) as well as a Gshift button that changes the behavior of all buttons on the mouse much the same way that a normal shift key changes the behavior of buttons on a key board. Due to this number pad, the mouse registers itself as a composite mouse and keyboard, so the computer is expecting to receive key input from the device.

Using USB PCAP [27], a tool which allows users to record USB traffic sent over the wire, and Wireshark, we were able to identify USB packets that corresponded to mouse movement and button presses. Four signals work in tandem to send a USB button press; see Table 1 for an example. HID related packets are prefixed with 0x01 and contain standard HID keyboard codes. Packets encoding the combination of buttons that are currently pressed begin with 0x80 and end with 0x14. Bytes 1 through 3 contain bit flags that represent all buttons currently pressed down, allowing for combos based off of pressing multiple buttons.

`0x 01 00 1E 00 00 00 00`	HID '1'
`0x 08 00 01 00 00 14`	G9 button pressed
`0x 01 00 00 00 00 00 00`	HID clear
`0x 01 00 00 00 00 00`	G9 button up

Table 1: The sequence of signals involved in sending an HID '1' over USB when button G9 is pressed

We also note that in the HID specification, modifier keys such as Alt, Shift, Control, etc. are sent as bit flags as well. Understanding how to send such keys plays a crucial role in our final exploits.

In the USB protocol, devices register various communication "endpoints" which can send and receive data. Each of these is numbered and can be set to fulfill a specific responsibility. The ATmega32u2 has 4 USB endpoints available for both input and output [1]. The output buffers can be populated with a fixed number of data bytes and flushed, triggering the data to be sent from the mouse down the USB cable. USB PCAP revealed that mouse movement was sent over USB endpoint 1 and button presses from the mouse are sent over USB endpoint 2. Endpoint 4 appears completely unused and is therefore an appealing candidate to use for communicating with malware running on a target machine.

The ATmega32u2 supports two approaches to sending consecutive USB packets: polling and interrupt driven. In the polling option, an application desiring to send data continuously checks the status of the RWAL (Read/Write Allowed) flag until it signals that the data bank is ready to be written to. In the interrupt driven approach, an interrupt is fired every time that a data bank is available to be written to. This interrupt is then caught by the appropriate handler. Our reverse engineering indicated that the mouse utilizes polling to determine when it can send USB packets; see §3 for more details.

2.4 The Mechanics of a Firmware Update

A firmware update to the mouse is initiated from the software updater on the host machine. A USB reset is sent to switch the mouse from the application segment to the bootloader segment, at which point the mouse begins listening for firmware update messages from the host. We

Firmware data	
Firmware data	$\texttt{0xc0}_0\ \texttt{00}_1\ \texttt{YY}_2\ \texttt{ZZ}_3\texttt{...ZZ}_{31}$
Firmware update control flow	
Start Update	$\texttt{0xd0}_0\ \texttt{00}_1\ \texttt{00}_2\ \texttt{01}_3\ \texttt{00}_4\texttt{...00}_{31}$
Start Page	$\texttt{0xd0}_0\ \texttt{WW}_1\ \texttt{WW}_2\ \texttt{02}_3\ \texttt{00}_4\texttt{...00}_{31}$
End Page	$\texttt{0xd0}_0\ \texttt{WW}_1\ \texttt{WW}_2\ \texttt{01}_3\ \texttt{00}_4\texttt{...00}_{31}$
End Update	$\texttt{0xd0}_0\ \texttt{00}_1\ \texttt{00}_2\ \texttt{03}_3\ \texttt{00}_4\texttt{...00}_{31}$

Table 2: The various signals involved in a firmware update. All packets are 32 bytes in length. For sending the firmware data the two-th byte indicates which data packet out of 8 it is. The rest of the packet contains actual program data. Bytes one and two of the Start Page and End Page signals contain the address the of page to write.

remind the reader that Logitech wrote a custom firmware updater in their bootloader, so there is no public documentation available.

There is a complex function in the bootloader that controls how it responds to various USB packets. For the purposes of this discussion, the only relevant packets are those that begin with 0xc0 which contain the actual bytes of the firmware and those that begin with 0xd0 which deal with various aspects of the control flow of the firmware update; see Table 2 for details. We gave the different packets semantic names based off of a series of "case statements" in the bootloader that switch off of various bytes of incoming USB packets.

Our analysis of the firmware update function in the mouse firmware revealed five kinds of packets: Start Update, Start Page, Firmware Data, End Page, and End Update. A Start Update packet causes the bootloader to erase all pages of program memory that contain application code in preparation for receipt of a new firmware version. The actual data transfer occurs page by page, where transferring one page involves one Start Page packet, eight Firmware Data packets, and then an End Page packet. Once all of the pages have been transferred, the host sends an End Update packet.

Upon receiving a Start Page packet, the mouse clears out a page-sized temporary buffer in program memory. Then, as the bootloader receives each Firmware Data packet, it writes the data into the temporary buffer. Upon receiving an End Page packet, the firmware updater writes the temporary buffer to program memory at the page address indicated in the packet.

3 Firmware Analysis

With standard reverse-engineering techniques, we were able to identify significant units of mouse firmware functionality. The firmware contains over 12,000 lines of assembly as well as 3 KB of data that contain information such as the version string and device name. Ultimately we gave precise, human readable names to over 160 functions compromised of over 4,000 lines of assembly. During this process we reversed engineered substantial portions of a USB library, an EEPROM library, an SPI library and a math library. We are happy to have avoided reversing the entire firmware; knowing what not to read is just as important as knowing what to read when attempting to quickly understand how a system works.

3.1 Obtaining the Firmware

Firmware for the Logitech G600 can be updated via software available from the Logitech website. Both 32- and 64-bit versions are available for download for recent Windows platforms [25].

Inspection of the firmware updater revealed that there were several Intel Hex (IHEX) files contained as Windows Resources within. With basic tools (Windows Resource Hacker [20]), we were able to extract the IHEX files — which we found to be different versions of the G600 firmware as AVR binaries — modify them, and replace them within the updater to be delivered to the mouse which we will describe in more detail in §4.

3.2 Analysis Process

We developed a workflow for analyzing the firmware with GCC tools. After extracting an IHEX file resource using Windows Resource Hacker [20], we converted the IHEX into an AVR binary file using `avr-objcopy`, then disassembled it for reverse-engineering using `avr-objdump`.

The manual was invaluable in understanding the sequence of steps necessary for reading from and writing to USB. We particularly benefited from the information it contained regarding various memory locations that are used for interacting with USB, EEPROM, and SPI. Identifying references to these locations allowed us to quickly pinpoint the leaf functions that dealt with the USB controller. Additionally, searching for the constants we saw in the PCAP (from §2.3) further helped us to locate functions responsible for sending USB HID codes.

We pair programmed heavily; it prevented many misunderstandings of the assembly. The merits of pair programming particularly shined each time we completed annotating a function and needed to articulate a higher level function name that captured the precise role of that function.

3.3 Static Analysis

To better focus our reverse engineering efforts, we wrote a static analysis tool capable of recursively extracting the call chain. This tool leveraged a symbol table that we updated throughout the reversing process so that the call chain used the meaningful names that we gave to functions rather than their raw addresses when possible. This helped us better visualize the overall flow of functions, allowing us to more easily understand higher level patterns in the firmware.

3.4 Hardware Debugging

We attempted to use microcontroller development hardware to aid us in the firmware analysis process. Tools like the AVR Dragon [2] allow users to use JTAG to set hardware breakpoints and to re-flash the microcontroller. However, we were unable to use JTAG with the Logitech G600. The ATmega32u2's manual indicates that irreversibly disabling JTAG is possible by blowing a hardware fuse, so we conclude this was Logitech's course of action. This is one of many best practices for mitigating attacks on embedded devices.

In addition to supporting JTAG, the ATmega32u2 supports debugWIRE, a proprietary hardware debugger offered by Atmel. However, we lacked the proprietary tools necessary to take advantage of debugWIRE and did not pursue it further.

4 Firmware Update Process

In this section we will discuss the steps we took to modify the firmware updater as well as the process we developed for packaging malicious firmware and deploying it to the mouse.

4.1 Modifying the Firmware Updater

On execution, the updater attempts to determine whether a G600 is connected to the computer. If a G600 is detected, it queries the G600 for its version of the firmware. If the version of the firmware running on the mouse is at least as recent as the version contained in the updater, the updater will inform the user that the mouse is up-to-date and exit. Otherwise, the user can initiate the firmware update process.

In order to guarantee that the modified firmware would reach the mouse, we used IDA Pro to examine the updater and identified the conditional branch which aborts the update process in the event of a failed version check. We patched the updater to make this branch an unconditional jump to the code path that initiates the firmware update.

Although Atmel offers a default firmware update protocol in the ATmega32u2 manual, Logitech elected to implement a custom protocol in their bootloader. Analysis of the bootloader section revealed that the mouse checks the firmware for corruption by calculating a CRC16-CCITT[36] as part of this protocol. We note that although the CRC is at the end of the application segment, it is not at the actual end of the firmware since the application segment is followed by the bootloader.

4.2 Preparing a Firmware Update

Once we identified the function responsible for sending HID codes, we hijacked the control flow of that function to additionally call a custom function that we placed in an unused section of program memory. After this initial change, we developed the following four step workflow to flash the mouse:

1. Write the custom functions in a stand alone assembly file

2. Run a script to modify the firmware and prepare it as an IHEX file

   (a) Assemble our custom functions using `avr-gcc`

   (b) Copy the appropriate bytes from the compiled binary to an unused portion of the original firmware

   (c) Compute a CRC of our modified firmware and add it to the firmware

   (d) Use `avr-objcopy` to convert the elf to an IHEX file ready to be injected into the mouse updater

3. Inject our modified firmware into the mouse updater using a Windows resource extractor

4. Flash the mouse by running the modified mouse updater

5 Results

The current state of the attack is that a modified Logitech firmware update utility first transfers new, arbitrary firmware to the mouse. The malicious firmware then waits for its internal timer to trigger and then sends arbitrary HID key codes in order to open a shell on the target machine and execute malicious commands. Following convention, we run `calc.exe`. Below we have included an alternative malicious command[2] that could be run if the computer has Internet access.

```
<WIN> + R
powershell.exe
Start-BitsTransfer
    -source http://pwn.com/pwn.exe
    -destination .\pwn.exe
.\pwn.exe
exit
```

If the target computer does not have Internet access, then the mouse needs to store and deliver the entire malware. To handle this scenario, we wrote a file transfer utility to copy a binary stored on the mouse to the host computer [3]. As a proof of concept, we transfered a binary representation of the Fibonacci sequence from the mouse to the host computer.

The mouse has over 6.75 KB of available space in the application section to store a malicious binary; we could squeeze in a slightly larger binary by storing a zipped version on the mouse and unzipping it after we've transfered it to the host computer. Despite the increases in malware size over the past several years, there are still many pieces of malware that fit into our available space [24].

It is worth noting that this exploit can trivially be turned into a VM escape by first sending the host key. Additionally, we observe that it is somewhat common for consoles to briefly pop up on a user's screen in Windows, often during an update. We videotaped our exploit, and the entire process takes less than one third of a second and is indiscernible from a console-based update process.

6 Other Triggers

There are several other options for triggering a firmware update including timers based off of user inactivity, communication with a adversary-controlled program on the target, and fingerprinting of the target.

6.1 Timers and Clandestine attacks

We have two 8 bit registers available to us to use a counter. So far, we have used that counter to count button presses so that we trigger the exploit every *n* button presses. This eases testing, but it is not subtle from a user's point of view. However, as mentioned above, the terminal only briefly flashes on the user's screen.

Instead of sending the exploit in response to user activity, it may be more subtle to respond to a period of user inactivity. We believe we can do this by modifying the main sleep loop, the code responsible for handling timer interrupts, and the code responsible for handling user-triggered interrupts such as mouse movement and button clicks. The counter would represent the length of time since the last user activity, and it would be reset to 0 every time the user triggers an interrupt. Every timer interrupt, we could increment the counter to give us a running tally of how long it has been since user activity. In the main sleep loop, we can check the counter and send the exploit if it has been sufficiently long. Although we have not actually implemented these enhancements, we have identified and annotated all of the relevant portions of the code.

If we need more than a 16 bit counter, it is possible to use the mouse's data space to store the current count. It was initially unclear what regions of memory were safe to overwrite due to the Harvard architecture of the mouse. However, during the reversing process we developed a fairly comprehensive view of the location in memory and purpose of the various global structs that store the mouse's state. If we were to avoid all such structs, trial and error might quickly reveal locations in data memory that we could reliably use.

In constructing any such timer, the adversary must wait long enough to ensure that the user is not looking while being careful to not wait so long as to let the target computer enter sleep mode or lock the screen.

6.2 Communicating With The Malware

An alternative approach to trigger the mouse's payload would be some sort of communication with the host machine. If the mouse was flashed by the target user's machine, then we can assume that there is some attacker-controlled code running on the machine. Since USB is not a restricted resource, the malware can send messages to the mouse in order to inform the mouse that the malware is already present and that the mouse need not send the payload. If the malicious mouse does not hear from the malware in a sufficient period of time, it can then trigger the payload and reinfect the host machine with the malware.

6.3 Making the Attack Targeted

It is conceivable that a USB device may in some way be able to fingerprint the host that it is connected to. A malicious USB device can report itself as various different devices, even a firewire-to-USB converter with direct memory access [5], opening various opportunities for gathering information about the host. If we can develop such a fingerprinting mechanism, we open up new avenues for targeted attacks. The malicious mouse can scan host devices for fingerprints until it determines that is connected to a target device and only then release its payload. This will allow us to infect the mouse on a less secure machine and avoid suspicion until we are connected to a target more secure machine. Alternatively, it would allow us to infect many mice but to only infect the specific hosts that we wished to target, decreasing the likelihood of detection.

7 Mitigations and Discussion

Logitech has taken responsible steps against the most trivial attacks. Disabling JTAG prevents attackers from gaining complete control over the mouse and bypassing intended update functionality in entirety. However, the firmware itself must be more carefully protected using digital signature verification.

7.1 Mitigations

Ideally, the mouse should only accept a firmware update that is authored by Logitech. In order to guarantee this, the mouse must authenticate the author of the code via a digital signature. We implemented a digital signature to evaluate the space requirements that this would impose. To be clear, we are describing a digital signature on the firmware itself, verified by the mouse before being executed, not a signature on the updater.

Although the mouse has several kilobytes of free space available, the digital signature verification code must fit in the bootloader since the bootloader controls the firmware update process. The bootloader segment has a maximum size of 4 KB on the ATmega32u2, and the Logitech bootloader uses 3.3 KB of that space.

We implemented PKCS#1 v1.5 signing and RSA in C, using SPONGENT/256/256/128 as our underlying hash function [21, 9]. We did so with a mind toward space efficiency and applied various compile-time flags to minimize the executable size. In total, the signature verification code took 1.5 KB: SPONGENT took 400 bytes and the rest of RSA and PKCS took 1100 bytes. Although this requires more space than is currently available in the Logitech bootloader, we do not view this as prohibitive in general on the ATmega32u2. Note that Balasch et al. were able to implement SPONGENT/256/256/128 using 12% less space with hand coded assembly [4], suggesting that there is room for compacting the code size even further.

However, this mitigation is not perfect. Due to the small amount of memory available, once a mouse has received illegitimate firmware, it has already overwritten its legitimate firmware with the unsafe version. This leads to the classic fail-open/fail-closed problem for the mouse, where the mouse must choose between running the untrusted firmware or completely ceasing to operate. In this case, ceasing to operate (failing closed) and waiting for a legitimate firmware update is probably the better choice, though it is still not ideal.

Exploitable vulnerabilities in the bootloader itself would allow the cryptographic verification to be bypassed, of course. We did not find any such vulnerabilities in our reverse engineering.

7.2 Limitations and Severity

Thus far, we have treated the Logitech G600 as a case study, discussing how it could be used to sabotage target users or corporations. However, since the G600 is a gaming mouse, this drastically reduces the population our specific exploit can reasonably target. This does not reduce the strength of our attack in general, as these techniques can be applied broadly as demonstrated by Netragard [29]. Nonetheless, the intended market for the mouse must be considered.

Competitive gaming and eSports have become increasingly popular over the past several years and the associated prize pools have grown accordingly. For example, the prize pool for the DOTA 2 international championships in July of 2014 was over \$10 million [35]. In general, such competitions are extremely concerned with cheating; players compete on airgapped systems that are constantly monitored for suspect behavior. Players are banned from bringing USB sticks (with configuration files, etc.) due to the fear that they will have exploits on the USB stick. However, players are allowed to bring their own mice, keyboards, and other peripherals since this is often an integral part of their performance. Our exploit makes it clear that even allowing players to bring their own peripherals opens the door for unfair play.

Another limitation of this style of exploit is that the reverse engineering process is laborious and often limited to a specific peripheral. Furthermore, there is great variation among mice in the general population. As such, this exploit is better suited for a targeted attack than mass exploitation. Such firmware update attacks are particularly viable at the individual or corporate level. The mice used within one company are often standardized, providing a high chance of success within a given company.

8 Conclusion

We have presented an end-to-end demonstration of malware hosting on an off-the-shelf mouse, the Logitech G600. We reverse-engineered the mouse firmware update process and showed how it could be subverted to install malware on the mouse. We showed how mouse-hosted malware could take advantage of the USB HID stack as an attack vector against the host. Contrary to the common belief that mice are benign, mice turn out to make an ideal vector for compromise of hardened or air-gapped computers.

We implemented cryptographic verification code required to prevent the mouse from loading illegitimate firmware in order to investigate the associated requirements. We conclude that signature verification code is not prohibitively large.

Cryptographic verification could be bypassed if there are exploitable vulnerabilities in the mouse bootloader itself, and an attacker with physical access to a mouse could modify its hardware or write malicious firmware directly to flash. Ultimately, host security against malicious mice must come down to distinguishing input (mouse movements and keystrokes) generated by the user from input faked by the mouse firmware, as proposed in DARPA's Active Authentication program [18].

Acknowledgements

Keaton Mowery and David Kohlbrenner aided us in the modification of the x86 Logitech firmware updater. Samuel Chen, who has extensive background in microcontroller software development, kindly worked with us on trying to enable JTAG with his personal AVR hardware and software. Michael Walter provided an excellent translation of the article from *c't*. Stefan Savage had helpful comments and suggestions.

References

[1] Atmel. Doc 7799: ATmega8U2/16U2/32U2 Complete. `http://www.atmel.com/images/doc7799.pdf`, September 2012. [Online; accessed 2014-03-12].

[2] Atmel. AVR Dragon Information Page. `http://www.atmel.com/tools/AVRDRAGON.aspx`, 2014. [Online; accessed 2014-03-10].

[3] Daniel Bachfeld. Im Auge der Maus. `http://www.heise.de/ct/heft/2014-8-Computermaeuse-laden-Schaedlinge-aus-dem-Netz-nach-2156334.html`, March 2014. [Online; accessed 2014-05-21].

[4] Josep Balasch, Bariş Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, et al. Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices. In *Smart Card Research and Advanced Applications*, pages 158–72. Springer, 2013.

[5] Michael "Becher, M. Dornseif, and C. N." Klein. FireWire: all your memory are belong to us. `http://cansecwest.com/core05/2005-firewire-cansecwest.pdf`, 2005. [Online; accessed 2014-03-12].

[6] Anthony Bellissimo, J. Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *Proceedings of HotSec'06*, pages 37–43, 2006.

[7] Brian Benchoff. A Real Malware In A Mouse. `http://hackaday.com/2014/03/31/a-real-malware-in-a-mouse/`, March 2014. [Online; accessed 2014-05-21].

[8] Brian Benchoff. Malware In A Mouse. `http://hackaday.com/2014/03/30/malware-in-a-mouse/`, March 2014. [Online; accessed 2014-05-21].

[9] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. SPONGENT: A Lightweight Hash Function. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 312–25. Springer, 2011.

[10] Peter Bright. US agency baffled by modern technology, destroys mice to get rid of viruses. `http://arstechnica.com/information-technology/2013/07/us-agency-baffled-by-modern-technology-destroys-mice-to-get-rid-of-viruses/`, July 2013. [Online; accessed 2014-03-13].

[11] Matthew Brocker and Stephen Checkoway. iSeeYou: Disabling the MacBook webcam indicator LED. In Kevin Fu, editor, *Proceedings of USENIX Security 2014*. USENIX, August 2012. To appear.

[12] K. Chen. Reversing and Exploiting an Apple Firmware Update. In *Proceedings of Black Hat 2009*, 2009.

[13] Stacy Cowley. Federal agency spent $3 million fighting non-existent malware. `http://money.cnn.com/2013/07/09/technology/security/commerce-malware`, July 2013. [Online; accessed 2014-03-13].

[14] Adrian Crenshaw. Plug and prey: Malicious USB devices. Presented at Shmoocon 2011, January 2011. Online: `http://www.irongeek.com/downloads/Malicious%20USB%20Devices.pdf`.

[15] Ang Cui, M. Costello, and S. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of NDSS Symposium 2013*, 2013.

[16] Jeroen "Sprite_tm" Domburg. Optical Mouse Cam. `http://spritesmods.com/?art=mouseeye`, December 2006. [Online; accessed 2014-05-21].

[17] M. Elkins. Hacking with Hardware: Introducing the Universal RF USB Keyboard Emulation Device: URFUKED. In *Defcon 18*, August 2010.

[18] Richard P. Guidorizzi. Security: Active authentication. *IT Professional*, 15(4):4–7, July/Aug. 2013.

[19] Indyaner. MEGA32U4. `https://imgur.com/a/uOhyn`, March 2014. [Online; accessed 2014-05-21].

[20] Angus Johnson. Resource Hacker. `http://www.angusj.com/resourcehacker/`, 2011. [Online; accessed 2014-03-10].

[21] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5, 1998.

[22] Franci Kapel. Convert Optical Mouse into Arduino Web Camera. `http://frenki.net/2013/12/convert-optical-mouse-into-arduino-web-camera/`, December 2013. [Online; accessed 2014-05-21].

[23] Jacob Kastrenakes. US Commerce Department destroyed $170,000 worth of TVs, mice, and more to root out malware. `http://www.theverge.com/2013/7/8/4503946/commerce-department-unnecessary-cybersecurity-computer-destruction`, July 2013. [Online; accessed 2014-03-13].

[24] Adam Kujawa. You Dirty RAT! Part 1 – DarkComet. `http://blog.malwarebytes.org/intelligence/2012/06/you-dirty-rat-part-1-darkcomet/`, June 2012. [Online; accessed 2014-05-22].

[25] Logitech. Logitech G600 Support and Downloads Website. `http://www.logitech.com/en-us/support/g600-mmo-gaming-mouse`, 2014. [Online; accessed 2014-03-10].

[26] Charlie Miller. Battery firmware hacking: Inside the innards of a smart battery. Presented at BlackHat 2011, August 2011. Online: https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf.

[27] Tomasz Moń. USBPcap. http://desowin.org/usbpcap/, Unknown. [Online; accessed 2014-03-10].

[28] Lawrence Munro. 418 I'm a teapot: Programmable HID USB Keystroke Dongles (PHUKD). http://418imateapot.blogspot.com/2010/11/programmable-hid-usb-keystroke-dongles.html, 2010. [Online; accessed 2014-03-10].

[29] Netragard. Netragard's Hacker Interface Device (HID). http://pentest.netragard.com/2011/06/24/netragards-hacker-interface-device-hid/, June 2011. [Online; accessed 2014-05-20].

[30] Jason" "Pisani, Paul" "Carugati, and Richard" "Rushing. USB-HID Hacker Interface Design. Presented at Black Hat 2010, 2010.

[31] Bruce Schneier. Dropped USB Sticks in Parking Lot as Actual Attack Vector. https://www.schneier.com/blog/archives/2012/07/dropped_usb_sti.html, July 2012. [Online; accessed 2014-03-12].

[32] USB.org sponsored by USB Implementers Forum Inc. USB HID Information. http://www.usb.org/developers/hidpage/, Unknown. [Online; accessed 2014-03-10].

[33] Symantec. Trojan.Mebroot. http://www.symantec.com/security_response/writeup.jsp?docid=2008-010718-3448-99, 2014. [Online; accessed 2014-03-10].

[34] Economic development administration: Malware infections on EDA's systems were overstated and the disruption of IT operations was unwarranted. Final Report No. OIG-13-027-A. Online: http://www.oig.doc.gov/OIGPublications/OIG-13-027-A.pdf, June 2013.

[35] Valve. The International - DOTA 2 Championships. http://www.dota2.com/international/compendium/, May 2014. [Online; accessed 2014-05-26].

[36] Ross Williams. A Paper on CRCs. http://www.ross.net/crc/crcpaper.html, August 1993. [Online; accessed 2014-03-12].

Notes

[1] The *c't* article makes a remarkable claim which we have not seen followed up on. Their off-the-shelf mice, when imaging mouse pads with a particular design, would send commands to the host that would download and install a remote-access tool. Patterns in the mouse pad encode the download trigger and the IP address to contact. They had received the mouse pads several months earlier as a free conference give-away.

[2] `Start-BitsTransfer` is the Powershell equivalent of `wget`. We note that the BitsTransfer utility may be used from `cmd` as well.

[3] We considered two options for writing the file transfer tool: `copy con` and Powershell. The `copy con` approach relies on using Alt codes while the Powershell approach would write the hex bytes into a Powershell byte array and then use `set-content` to write to file. We implemented the `copy con` approach and were disappointed to discover a flaw: 0x00 terminates the transfer. The Powershell approach is capable of writing all bit patterns.