Practical attacks on the i>clicker classroom response system

Thomas Hebb
Tufts University

Dec 29, 2016

Abstract

As embedded computing technology has become more affordable, a number of products have emerged that attempt to enhance traditional lecture-based academic courses by giving instructors digital means of soliciting student participation. One such product is the i>clicker, which allows instructors to pose multiple-choice questions during a lecture and see the responses of students in real time. In this paper, I present the results of a security audit of the protocols and software that make up the i>clicker system, with special focus to attacks that can be carried out by an average student without access to specialized hardware.

1 Introduction

In recent years, professors at a number of colleges and universities—including at Tufts—have integrated classroom response systems into their lectures. These systems enable instructors to collect and aggregate students' responses to questions in real time. Each student purchases an RF remote, often called a "clicker," which has, at minimum, buttons marked A through E which the student can use to privately respond to multiple-choice questions. The clickers report responses over radio to a USB base station, which in turn sends the responses to software running on the instructor's computer. The software allows the instructor to aggregate and display responses of everyone in a class, associate students with their clickers, and optionally track attendance and responses for grading purposes.

When used effectively, clickers can benefit both a professor and their students. Consider a professor who writes a problem on the board and wants to gauge how well their students understand how to solve it. In a traditional lecture, one or two students might raise their hands and answer. But students who volunteer answers are likely to be those with the best understanding of the problem. To judge the class' understanding as a whole, the professor ideally wants to collect an answers from all the students. One way to do this is to present a set of possible answers and ask for a show of hands for each choice. But this method makes each student's answer public knowledge and can encourage students to go with the majority opinion even if they disagree with or don't understand it. Clickers solve the problem by allowing the professor to collect responses privately and display them in an aggregate, anonymized form. Some professors keep responses entirely anonymous, using them only to gauge understanding, while others count the presence or correctness of a student's responses towards that student's grade.

It should be clear why, in the latter case, the security model of a classroom response system is an important concern. A system with poor attribution or confidentiality guarantees can be exploited by a student to improve their own grade (by sniffing and copying correct answers of others) or to sabotage the grades of others (by transmitting spoofed, incorrect answers attributed to a victim). Prior research into the i>clicker system—one of the most popular classroom response systems and the focus of this paper—has shown its protocol to be entirely unauthenticated, allowing those with sufficient hardware to indiscrim-

inately sniff and spoof responses [2]. However, the hardware required to do this is nontrivial: one must have either a clicker unit and means to reprogram it or a generic, reprogrammable radio that can receive and transmit on the frequencies used by i>clicker remotes. The time and effort needed to obtain and configure this hardware is likely not worth the small grade improvement a malicious student could obtain, especially since clicker responses generally make up a small ($< 10\%$) portion of a course's overall grade.

One might assume that, in courses where students' clicker responses don't affect their grades at all, the security of the clicker system is irrelevant since students have no motivation to sniff or spoof clicker responses. This assumption is incorrect. In this paper, I present two new vulnerabilities in the i>clicker system, both of which can be exploited by a student with only a laptop computer and used to gain remote code execution on the instructor's computer. Additionally, I provide a proof-of-concept exploit for one of the vulnerabilities.

2 To the Community

I chose this topic to serve as a case study of how the most prominent security vulnerabilities in a system are often not the most serious. In the case of clickers, it's natural to assume that sniffing and spoofing students' responses is the most an attacker can do after breaking the system. However, as I show, flaws in the implementation of the i>clicker application software expose much higher-value vulnerabilities to attackers—vulnerabilities which have nothing at all to do with the intended purpose of the system. Specifically, vulnerabilities are present which make remote code execution much easier. Thus, the introduction of i>clicker into a network allows attackers to gain control of previously-secure computers, potentially using the compromised systems as footholds to exfiltrate information or leverage previously-unexploitable vulnerabilities to gain even higher privilege levels.

Cases like these highlight the need both for improved security auditing of vendors by corporate and academic IT departments and for application-level sandboxing at the OS level: a compromised application should not be able to read arbitrary data belonging to users who run it, nor should it be able to spread itself to network mounts and removable drives that it doesn't normally require access to; yet on modern Windows and Linux systems, both are all too possible.

3 Discoveries

The i>clicker system consists of three components: the remotes (clickers), the base station with which the remotes communicate over RF, and the application software with which the base station communicates over USB.

To audit each of these components, I took a bottom-up approach: I started by analyzing what happens at the lowest layer of each system, and used that knowledge to figure out what the higher layers do. For each hardware component, this meant removing the casing from a unit and identifying the microcontrollers powering it, finding those microcontrollers' datasheets, and, if possible, extracting the programs running on those microcontrollers. For the application software, this meant running it on an open operating system (Linux) and analyzing the network and USB traffic it generated.

The clicker units turned out to have almost no attack surface. The units are entirely self-contained, and the only way to reprogram one is to disassemble it and connect a specialized programmer to the programming port (Figure 1). Even with a programmer, the existing software on the unit can't be read due to security features integrated into the microcontroller[1], so an attacker would have to write entirely new software for the unit by tracing connections on the PCB and referencing publicly-available datasheets. This is a significant amount of work for not much reward, as all a compromised clicker can affect is the system's radio communications. As previously discussed, sniffing and spoofing radio communications gives an attacker little to no benefit in most situations.

[1] This was not the case for the (now-obsolete) hardware version studied by Gourlay et al. [2].

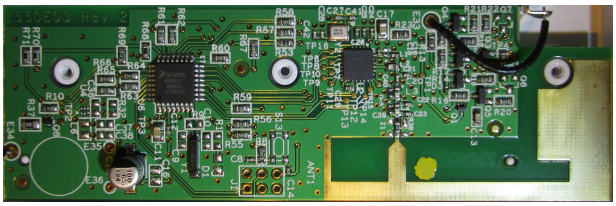

Figure 1: i>clicker+ PCB, showing NXP MC9S08QE8 microcontroller (left) and programming port (center bottom)

The other two components of the system—the base station and the application software—were much more promising. The base station connects to a computer over USB, which means that an attack can conceivably be carried out with no specialized hardware. As it turns out, the base station exposes an unprotected USB firmware update mechanism, so any attacker who gains brief physical access to a base station can reprogram it with malicious firmware which attempts to infect any computer subsequently connected to that base station. This attack is described in Section 3.2.

The application software is the most high-value target, as an attacker who compromises it immediately gains remote code execution. Unfortunately, this sensitivity is not reflected in the software's implementation: the software also falls to a poorly-implemented update mechanism. Any attacker who can intercept a victim's network traffic—i.e. any attacker on the same network as their victim—can supply a forged software update and plant arbitrary code on the victim's computer. This attack is described in Section 3.1.

3.1 Forged software update

The i>clicker software is capable of updating itself. The update process is initiated by a user clicking "Check for Update" in the software's Help menu. Once begun, the process is vulnerable to a man-in-the-middle attack: the software uses HTTP to download an XML file which indicates if an update is available. If one is, the XML file contains the URL of a zip (for Windows) or tar (for Linux and macOS) archive containing the update. The software then downloads that archive, checks its size and checksum against values from the XML file, and, if they match, replaces the existing installation with its contents. As HTTP is an unauthenticated protocol, an attacker can intercept the software's request for the XML file and provide a forged file pointing to an archive that contains a malicious payload. If the payload is contained in the application's main executable, it will automatically run when the software restarts itself after updating.

For an attacker to exploit this vulnerability using traditional man-in-the-middle methods (ARP and DNS spoofing), they must be on the same network segment as the victim. This is a reasonable assumption to make, as any student attempting to attack an instructor's computer will almost certainly be connected to the same wireless network as the instructor during the lectures where i>clickers are used.

I had originally planned to use the existing Evilgrade [1] framework to implement a proof-of-concept exploit for this vulnerability. However, I found the framework to be lacking in documentation and have poor core code quality. Additionally, it couldn't repack archives on-the-fly, so I would have had to hardcode a payload for the exploit. Because of these limitations, I instead wrote my own Python script [3] to exploit the vulnerability. My script runs a web server which emulates the real i>clicker update server. By rredirecting requests to `update.iclickergo.com:80` to the script, an attacker can install a malicious payload on their victim's computer.

Admittedly, this attack is hampered by the fact that it requires user action—the operator of the i>clicker software must click the "Check for Update" button and agree to the update. However, with a bit of social engineering, this is only a small obstacle. In the situation where the attacker is a student conducting a man-in-the-middle attack during a class, the student can (1) stop forwarding packets, (2) wait for the inevitable symptoms of a missing internet connection to appear, and (3) interject with a helpful comment to the effect of "Excuse me! This happened in my last class, and updating the i>clicker software fixed it." As most school IT departments strongly en-

courage students and instructors to keep their software patched and up-to-date, most instructors will probably attempt an update without suspicion. Once the malicious payload is installed, the student can begin forwarding packets again. From the instructor's point of view, the update completed successfully and had the expected effect. They likely won't give it a second thought.

3.2 Malicious base station firmware

Although it involves hardware instead of software, this attack is actually quite similar to the previous one. The i>clicker base station contains two microcontrollers, shown in Figure 2: the first is an Intel 8052 microcontroller contained within the USB controller IC, which manages USB communications and the base's LCD display; the second is an Atmel AVR microcontroller, which manages radio communications. i>clicker provides a utility to update the firmware of the base station. By analyzing the network traffic of this utility, I found that it downloads unsigned, unencrypted firmware images for both microcontrollers. Unfortunately, another man-in-the-middle attack is likely infeasible, as the base station updater is a separate program which most instructors won't bother installing. However, the fact that the firmware images aren't authenticated in any way suggests another attack: an attacker with physical access can use the same USB protocol that the firmware update utility uses to install malicious firmware on the USB controller IC. The IC can be configured in firmware to present itself as any type of USB device [6]. An attacker could, for instance, make it emulate a USB keyboard, send keystrokes to launch a terminal, and run arbitrary commands on any computer it's connected to. There is significant prior work that further develops attacks possible with USB access to a victim's computer—Kamkar [4], for example.

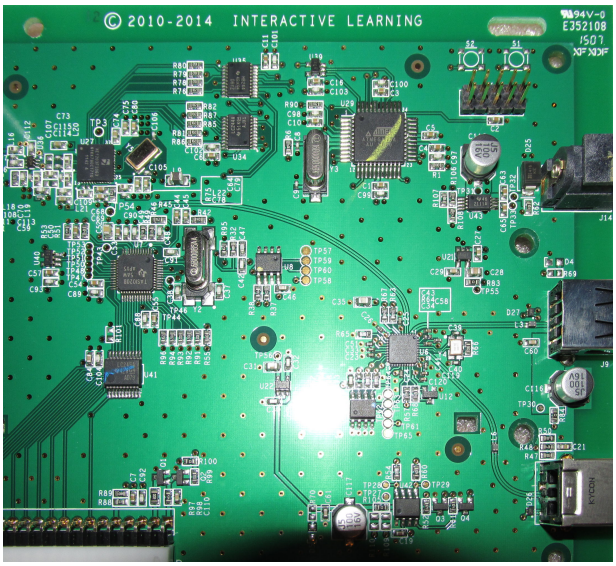

Figure 2: i>clicker Base PCB, showing TI TAS1020B USB controller IC (left center), associated firmware EEPROM (center), and Atmel ATMEGA16A (top)

4 Defenses

Both of these attacks can be prevented by someone who's aware of their existence: a savvy user can ensure that they never run updates on an untrusted network and that they never forfeit physical control of their base station. However, as I said in Section 2, the underlying issue is that software is often deployed without being audited at even a cursory level for security vulnerabilities: if IT departments more thoroughly screened technology used on their networks, the two vulnerabilities I found would likely have been fixed (or at least mitigated) a long time ago.

These attacks also show security weaknesses of modern desktop operating systems: the first attack, for example, works only if the target machine will execute the malicious payload installed by the flawed update process and allow it to access or modify sensitive data. Modern Windows and Linux systems, alas, allow both of these to happen: no code signing or sandboxing is enforced for processes running within a single user session without elevated privileges. So a malicious binary can access and modify every file belonging to the compromised user—including files on mounted network shares and session cookies for websites saved by the user's browser. I suspect that the situation is better on macOS, which by default

requires applications to be signed. Default security policies such as these are crucial to limiting the damage that can be caused by poorly-written and insecure software.

The second attack is harder to mitigate in a generic way: USB was designed as a single protocol for a wide variety of different devices; as such, USB devices are allowed to expose any functionality they desire. One way operating systems can prevent malicious devices from silently taking control is by prompting the user to authorize each device upon first connection. This brings its own set of problems, however: for one, an exception would have to be made for input devices, as a user can't respond to the prompt for a mouse or keyboard without being able to use that mouse or keyboard! Additionally, users would likely just accept the prompts without reading them, as infamously happened with Windows Vista's UAC prompts.

USB-based attacks have entered the spotlight in recent years, notably in the form of BadUSB [5], which refers to a set of vulnerabilities in USB flash drive controller chips that let attackers reprogram them to act as different device types—just like attackers can reprogram the i>clicker base.

5 Summary

In this paper, I presented two new attacks on the i>clicker classroom response system, each allowing a malicious student in a course to achieve remote code execution on their instructor's computer. Neither attack used any novel techniques; had the i>clicker software been engineered with security in mind, neither exploitable vulnerability should have existed. These vulnerabilities highlight the need for proprietary products to undergo security audits before being deployed on computers with access to sensitive information and also show the security benefits of OS-level sandboxing and signature verification.

References

[1] Francisco Amato. *Evilgrade*. GitHub. Software. URL: `https://github.com/infobyte/evilgrade` (visited on 12/17/2016).

[2] Derek Gourlay et al. *Security Analysis of the i>clicker Audience Response System*. Term Project Report. University of British Columbia, Dec. 7, 2010. URL: `https://courses.ece.ubc.ca/cpen442/term_project/reports/2010/iclicker.pdf` (visited on 12/19/2016).

[3] Thomas Hebb. *Clickjacker*. GitHub. Software. URL: `https://github.com/tchebb/clickjacker` (visited on 12/29/2016).

[4] Samy Kamkar. *USBdriveby*. Dec. 17, 2014. URL: `http://samy.pl/usbdriveby/` (visited on 12/20/2016).

[5] Karsten Nohl, Sascha Krißler, and Jakob Lell. *BadUSB. On accessories that turn evil*. Talk at PacSec 2014. URL: `https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf` (visited on 12/20/2016).

[6] *TAS1020B USB Streaming Controller Data Manual*. Texas Instruments. May 2011. 119 pp. URL: `http://www.ti.com/lit/gpn/tas1020b` (visited on 10/11/2016).