

Security Analysis of an IP Phone: Cisco 7960G

Italo Dacosta, Neel Mehta, Evan Metrock, and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology
{idacosta,giffin}@cc.gatech.edu
{nmehta, evan.metrock}@gatech.edu

Abstract. IP phones are an essential component of any VoIP infrastructure. The hardware constraints and newness of these devices, as compared to mature desktop or server systems, lead to software development focused primarily on features and functionality rather than security and dependability. While several automated tools exist to test the security of IP phones, these tools have limitations and can not provide a strong guarantee that a particular IP phone is secure.

Our work evaluates the attack resilience of a widely deployed IP phone, the Cisco 7960G, employing techniques such as: vulnerability scans, fuzz tests, and static binary analysis. While the first two techniques found no vulnerabilities, the static analysis of the firmware image revealed critical vulnerabilities and fundamental software design flaws. We conclude that security designs proven useful in desktop and server software architectures should similarly appear as part of the software design for devices such as IP phones.

Key words: VoIP security, IP phone, static binary analysis, embedded system security

1 Introduction

Voice over IP (VoIP) is changing the nature of physical telephones. For more than a century [1], a telephone endpoint consisted of a transmitter and receiver connected to the public switched telephone network (PSTN) via twisted pair wiring. In contrast, VoIP handsets, or *hard phones*, are connected to data networks via broadband Ethernet cables and configured with an IP address. They execute operating systems and application software on general purpose device-class computer hardware. Their architecture is closer in design to desktop computer systems than to PSTN-connected telephones, and as a result, they introduce the security weaknesses of the computing world into telephony. Telephones can now be targets of attack and entry points into an enterprise's internal network, concepts unfamiliar in the telephone network of the past.

Security is one of the main challenges of VoIP systems, and the security of the IP phones is a part of this challenge. The widespread deployment of these devices in organizations and the possible persons that could be using them—company executives, congressmen and -women, and so on—make IP phones an interesting target for attackers. The implications of a compromised IP phone



Fig. 1. Cisco IP Phone 7960G

could be enormous (e.g. the Greek Watergate case [2]): from simple denial of service attacks to eavesdropping of confidential information, unauthorized use of the device, and financial losses (e.g. unauthorized international calls).

In this paper, we analyze the security of the software executing on a widely deployed IP phone: the Cisco 7960G (Figure 1), part of the Cisco Unified IP Phone series. Prior to our research, the IP phone had a good security record of only several non-critical software flaws. Our analysis included a mix of runtime execution analysis using off-the-shelf fuzz testing software for implementations of the session initiation protocol (SIP), as well as static analysis of the binary firmware image stored in the read-only memory (ROM) of the IP phone. Fuzz testing revealed no flaws in Cisco’s software, which we hypothesize indicates that Cisco has included similar fuzz testing as part of their software development and quality assurance procedures. Our static code analysis, however, yielded a troubling conclusion: the software contained buffer overflows and other implementation errors that would allow a remote attacker to gain full control of the IP phone. Of greater concern is architectural design weaknesses in the operating system, which provides no memory protection among applications or between applications and the kernel.

IP phone software development seems poised to repeat the security errors that occurred during past desktop and server software development. By our analysis, the buffer overflow vulnerabilities existed in the Cisco phone because the firmware image contained calls to functions similar to known flawed functions, like `strcpy`, used in desktop and server systems. The software architecture appears to be evolving in a manner similar to desktop operating system development over the past 30 years. The maturity of the IP phone’s current operating system resembles that of real-mode operating systems, such as DOS. Unfortunately, this new software evolution is occurring in a networked environment much different than that present during the development of desktop systems, and IP

phone developers should recognize the risks of today’s networks while architecting these network-connected systems. Although this paper focuses specifically on the Cisco 7960G firmware, we have performed preliminary studies into the security quality of other devices, including a Siemens IP phone and a Windows Mobile installation on a mobile handheld. Our conclusions drawn from the Cisco firmware analysis indeed appear representative of security weaknesses in these other systems.

Our threat model includes an attacker that gained access to a corporate network where the 7960G has been deployed, and the expectation that the attacker is able to reach IP phones through this network. The ability to send input to the IP phone could be available to an attacker in the Internet at large, although some network configurations may limit this to attackers on the internal LAN. The hardware needed for a typical attacker to write and test exploit code is inexpensive and readily available. The access to an IP phone would likely be as simple as purchasing one from Cisco, a reseller, or an auction site. We assumed that the IP phones are configured with basic SIP functionality and that the IP phone administrator has taken basic steps to limit trivial historical attacks such as supplying compromised firmware using a TFTP server [3]. Cisco today distributes signed firmware images that limit the efficacy of these legacy attacks.

The Cisco 7960G uses an embedded ARM processor which makes the analysis of the firmware image challenging. Previous researchers have used static binary analysis to understand the execution of x86 or SPARC executables, but unique properties of ARM processors, such as the support of multiple instruction sets, introduce complications to the static analysis of IP phone firmware. However, a knowledgeable attacker or analyst should be able to work through the difficulties to learn security properties of the IP phone. We present the static binary analysis techniques that we used to uncover security vulnerabilities in the firmware with the expectation that the techniques may prove instructive to other security analysts intending to improve an IP phone’s security.

Our study produced the following contributions and results:

- **Limitations of automated security testing.** Our first set of tests used automated vulnerability scanning and fuzz testing tools in an attempt to trigger faults in the IP phone, but found no flaws (Section 4). In addition, published security advisories regarding this device described vulnerabilities discovered using automated tools but none of the vulnerabilities found by our in-depth static analysis. These facts show that the IP phone likely has been previously analyzed with automated tools but that critical vulnerabilities still exist.
- **Vulnerability discovery.** Using static binary analysis, we discovered fundamental architectural weaknesses in the design of the firmware, three stack buffer overflow vulnerabilities, and one heap overflow vulnerability (Section 5). The most critical vulnerability could be exploited by a remote attacker with a single network packet and would result in full device access.
- **Suggested architectural changes and other recommendations to improve security.** We examined security measures that were present or

absent on this embedded device and compared this feature set to modern non-embedded systems. Combined with our security assessment, we were then able to suggest patches for vulnerabilities and other improvements that could make the platform more secure. These included architectural changes, changes to compilers, changes to APIs, and further security review by the embedded device vendor.

This paper contains descriptions of the vulnerabilities found by our security analysis, but we have deliberately delayed publication of this paper so that the details would no longer be relevant to the current firmware deployed on Cisco IP phones. We performed our analysis during spring months of 2007 and reported to Cisco the vulnerabilities discovered by our research in the summer of 2007. In February 2008, Cisco publicly disclosed the existence of our critical vulnerabilities and released patched firmware for the affected IP phones in the Unified IP Phone series [4].

The next section provides background information on IP phones and previous commentary regarding their security. Section 3 describes the experimental testbed used for our runtime tests. Section 4 presents those automated tests performed against the IP phone and the test results. Section 5 describes the methodology and challenges faced during the disassembly of the binary firmware image and the results of the static binary analysis, including examples of the vulnerabilities found and our recommendations. Finally, Section 6 contains related research work in the area of VoIP vulnerability analysis and embedded system security, and in Section 7 we present our conclusions.

2 Background

This section provides background information on IP phones that knowledgeable readers may choose to omit.

IP phones can be classified in two types: hard phones and soft phones. Hard phones are typically embedded devices with a real time operating system (firmware), applications, and physical appearance similar to traditional telephone handsets. In comparison, soft phones are software applications that run on general-purpose desktop computers and handle voice traffic using the standard computer speakers and microphone (or any other specialized equipment connected to the computer). IP hard phones from companies such as Cisco, Siemens, and Avaya are widely deployed in many business organizations, and soft phone applications such as Skype and Google Talk are very popular on desktop computers in business and home environments. By 2005, Cisco sold more than 6 million IP hard phones [5] belonging to same product family of the device studied in this paper, the Cisco 7960G.

In general, soft phones are considered less secure than hard phones because soft phones inherit the vulnerabilities of the operating system where they run. For example, any malicious software, such as a virus, worm, trojan horse, or bot, that affects a vulnerable computer can also affect the soft phone applications

running on that computer. Due to such attack potential, NIST's *Security Considerations for Voice over IP systems* [6] recommends that soft phones should not be used where security or privacy are a concern.

On the other hand, hard phones (henceforth referred to as IP phones) are considered more secure because they run a smaller and simpler operating system with few applications and services: they are embedded devices with limited hardware resources. For example, a common metric used to measure the security quality of code is the number of bugs per thousand lines of code (KLOC). This varies from system to system, but public estimates document bug densities in the range of 5 to 50 bugs per KLOC [7]. While today's desktop operating systems consist of millions of lines of code, embedded operating systems consist of only thousands of lines. Microsoft Windows Vista requires 15 GB of available space for installation while the firmware for the IP phone used in this paper requires around 1 MB of space. Given this comparison, it is easy to assume that IP phones and embedded devices in general have fewer vulnerabilities.

However, the resource constraints of the IP phones can also have a negative impact on their security. In most cases, the user of an embedded system does not have the ability to add security applications (i.e. antivirus, firewalls, etc.) that were not included by the manufacturer, and it can be difficult to apply security patches or firmware upgrades. Moreover, Raghunathan et al. [8] and Kocher et al. [9] described how the unique characteristics of embedded systems such as IP phones made it difficult to implement effective security in their software.

The 7960G has a good security history with only five reported vulnerabilities [10] between 2003 and the time of our analysis in spring 2007: four denial of service (DoS) vulnerabilities and one security bypass vulnerability due to a design flaw in its TFTP-based firmware distribution method. Arkin [3] pointed out flaws related to the deployment and supporting environment of the Cisco IP phones, particularly in the use of TFTP to configure the devices. Today, firmware files are digitally signed by Cisco to protect their integrity and the files are authenticated against a certificate trust list to limit an attacker's ability to tamper with the files in transit on the network. It is still up to the user to properly configure certificate trust lists and sign configuration files before they are distributed across a network. In particular, without a full Cisco Call Manager deployment, it can be very challenging to implement this hardening feature. There are no publicly documented numbers that state the percentage of Cisco IP phone real-world deployments using these security features.

Several automated security tools such as vulnerability scanners and fuzzers have been developed to test for VoIP vulnerabilities in the different elements of a VoIP system, including IP phones. A comprehensive list of such tools is maintained by the Voice over IP Security Alliance (VOIPSA) [11]. These tools have helped academic and industry security researchers find several critical vulnerabilities in IP phones, resulting in security advisories, new firmware versions, and security patches. For example, the Madynes research team at INRIA Lorraine reported several vulnerabilities in IP phones using their stateful SIP fuzzer [12].

However, the use of these tools is limited because they can not find or test for all the possible vulnerabilities in a given device. Therefore, one must be careful with the results obtained by running these automated tools. In this paper we show how static binary analysis techniques similar to those used for desktop operating systems and applications can be used to find critical vulnerabilities that have not been identified by the automated tools.

3 Experimental Design

In our tests, we used a Cisco IP phone 7960G (Figure 1), one of the most popular models of the Cisco Unified IP Phones Series 7900. This is a flexible device that supports three VoIP protocols according to the firmware image loaded: skinny client control protocol (SCCP), media gateway control protocol (MGCP), and session initiation protocol (SIP). The 7960G also supports several network protocols (DHCP, NTP, TFTP, and LDAP3), audio compression codecs (G.711 and G.729a), compatibility with H.323 and Microsoft NetMeeting, and XML services support.

We analyzed version 8.2 of the free SIP firmware image available from Cisco’s website. The firmware is distributed as a compressed ZIP archive containing 5 files, described in Table 1. As mentioned in Section 2, the comparison of the size of the firmware files with the size of today’s general purpose operating systems (kilobytes vs. gigabytes) provides an idea of the complexity of the software applications running in the IP phone and the probability of finding bugs in the firmware image.

Table 1. SIP Flash Image for 7940/7960 IP Phone v8.2 (0)—Non CallManager

File Name	Size	Description
OS79XX.TXT	14 bytes	Contains the universal application loader image name
P0S3-08-2-00.loads	461 bytes	Contains the universal application loader and application image names
P003-08-2-00.bin	129,240 bytes	Non-secure universal application loader for upgrades from pre-5.x images
P003-08-2-00.sbn	129,644 bytes	Secure (digitally signed) universal application loader for upgrades from pre-5.x.images
P0S3-08-2-00.sb2	785,338 bytes	Application firmware image

To transfer the SIP firmware image to the IP phone, we set up a TFTP server with the files described in Table 1. We included an additional configuration file, `SIPDefault.cnf`, which configured most of the IP phone’s parameters, such as network information, SIP configuration, and security parameters. For more details about the IP phone initialization and configuration options, see the Cisco SIP IP Administrator Guide version 8.0 [13].

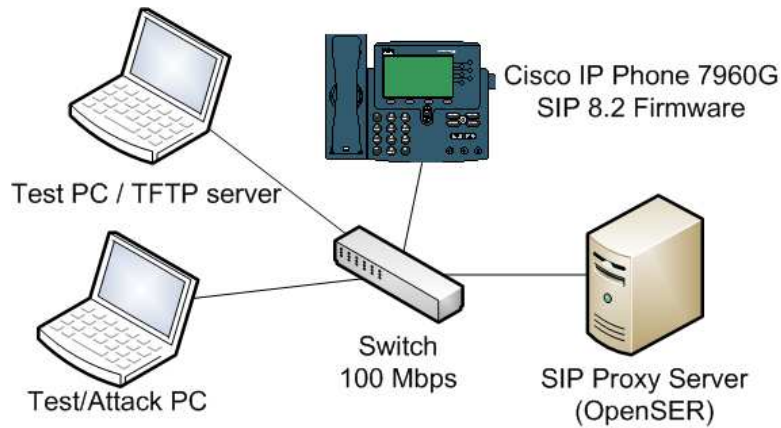


Fig. 2. Experimental Testbed.

After loading the SIP firmware in the IP phone, we configured a SIP proxy server (OpenSER v1.1.1) to provide a basic SIP environment for the IP phone. A SIP proxy server helps route requests to the user’s current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users [14]. We configured and tested the IP phone functionality establishing VoIP calls between the IP phone and a SIP softphone (3CX IP Phone) installed in a test PC. The resulting infrastructure is depicted in Figure 2.

4 Runtime Analysis

To test the IP phone’s runtime behavior, we performed both vulnerability scans and fuzz tests of the device. For our vulnerability scans, we connected to the IP phone from our test machines, established a variety of malformed connections, and observed the IP phone’s responses. We used the Nmap network scanner to detect listening network ports on the IP phone. Nmap only detected two open ports: 123/UDP (NTP) and 5060/UDP (SIP). The scanner was also able to identify that we were scanning a Cisco device. After the network scan, we ran a vulnerability scan against the IP phone using the Nessus vulnerability scanner with all attack plugins activated (12,239 plugins). The vulnerability scan did not reveal any security-relevant results, and the IP phone continued operating normally during and after the scan. However, we expected this result because vulnerability scanners can only detect previously discovered vulnerabilities via the use of attack signatures (Nessus’ plugins).

Fuzz tests send random and malformed input to software to quickly assess its security quality [15]. We used open-source SIP and RTP protocol aware fuzzers to test the IP phone for commonly made coding mistakes that may reveal vulnerabilities. Specifically, we were looking for signs of instability, such as device

resets or high CPU utilization, when exposed to unexpected input. Any sign of instability in the IP phone would have implications for device availability, and could lead us to significant vulnerabilities in the software application. We chose four fuzzing tools based on the VoIP Security Alliance (VoIPSA) VoIP Security Tool List [11] (Table 2).

Table 2. Scanning and fuzzing tools used.

Name	Type	Comment
Nmap	Network Scanner	65,535 TCP/UDP ports scanned
Nessus	Vulnerability scanner	12,239 vulnerability plugins tested
PROTOS	SIP Fuzzing tool	4,527 SIP INVITE test messages
ASTEROID	SIP Fuzzing tool	Approximately 36,000 SIP test messages
SIP Forum Test Framework	SIP Fuzzing tool	Approximately 70 SIP tests
Ohrwurm	RTP Fuzzing tool	RTP payload is fuzzed with a constant bit error ratio (BER)

PROTOS is a framework for testing implementations of protocols using a black-box approach. It has several test suites for testing different protocols, including HTTP, DNS, and LDAP. For our tests, we used the SIP test-suite (c07-sip), which consists of 4,527 maliciously modified SIP INVITE messages. Test messages are grouped by the type of vulnerability that they try to exploit: buffer overflows, format string attacks, malformed sequences, and others. However, PROTOS is stateless and therefore can only test for attacks that required just one message to be sent to the targeted device. A further disadvantage is that it only tests INVITE messages, therefore, possible vulnerabilities associated with other SIP messages will not be found. While it is possible to add more test cases, there is no automated way to do so.

Asteroid is a more rudimentary fuzzing tool. It has around 36,000 crafted messages and it can test several types of SIP messages, such as INVITE, REGISTER, BYE, CANCEL, OPTION, SUBSCRIBE, and NOTIFY. However, the test messages are prepared less carefully (i.e. random data), and it also stateless and does not track the status of the SIP sessions.

SIP Forum Test Framework (SFTF) is a testing framework to test different SIP elements for common errors in protocol implementations. This tool uses more knowledge about the semantics of the SIP sessions, and it can test more advanced scenarios, including the use of registration and DNS. The number of test cases included in the distribution package is small (around 70 tests) when compared with the previous tools. As with other tools, SFTF has a flexible design that supports the addition of new test cases, although there is no automated way to generate tests.

Ohrwurm is a small and simple real time protocol (RTP) fuzzer. To use it, an existing media session must exist between two IP phones, and a man-in-the-

middle attack is necessary to access the RTP streams between the end-points if a switched LAN is used for the test. The tool reads the sequence numbers of the RTP streams and injects noise with a constant bit error ratio (BER).

We ran each tool multiple times against the IP phone and monitored the phone’s behavior in response to the specially crafted malicious input streams generated by the fuzzing tools. We did not notice any abnormal behavior on the IP phone during our testing. In an unexpected result, the soft phone used to establish an RTP session with the IP phone while using Ohrwurm repeatedly stopped responding.

The results of the runtime tests were unsurprising because previous studies exposed the IP phone to a wide variety of vulnerability scanners and fuzzing tools. Many of the previous publicly-announced security issues with this device were indeed found through fuzz testing. For example, the latest reported vulnerability against this IP phone model was a denial of service vulnerability found by the Madynes VoIP fuzzer [16]. One might assume that because fuzz testing did not reveal any security vulnerabilities, we could conclude that the Cisco 7960G IP phone is very secure. However, the additional perspective gained from in-depth static binary analysis provided us with a better understanding of the platform and a different conclusion.

5 Static Analysis

We statically analyzed the binary firmware of the Cisco 7960G to identify coding flaws that could lead to security exploits against the IP phone. This analysis occurred in three steps. First, we disassembled the binary image to obtain the ARM assembly code of the firmware. Second, we manually analyzed the control flow and data flow of the code to understand how the software executes. Finally, we applied security analyses to identify fundamental design flaws and four coding vulnerabilities that weaken the IP phone.

5.1 Firmware Disassembly

The first step in disassembling the device was to determine the underlying processor architecture for the operating system and applications running on the IP phone. This allowed us to infer other attributes about the code we were examining, such as the split between code and data and the absolute virtual addresses where portions of the file would be mapped.

As stated in Section 3, the firmware contains several files within a compressed archive (Table 1). Each application on the device is denoted by an 8-byte identifier easily located within the file header. It is possible to have multiple applications within one discrete file. Much of the code that was security-relevant and operated on externally-supplied data was located in the file ending with the extension `.sb2`. The `.sb2` file can be subdivided into two distinct applications, one of which, “PAS3ARM1”, contained the basic components of a real-time operating system responsible for maintaining operation of the IP phone and processing any network I/O.

We identified “PAS3ARM1” as interesting for security analysis. In the firmware version examined in our research, the digital signature comprised the first 404 bytes of the file. After this has been manually removed, the beginning of the application data was found. A quick disassembly with IDA Pro, a common commercial disassembler, revealed that the application contains ARM processor code. We could locate distinct functions with correct function prologues and epilogues. The identification and confirmation of the processor architecture was the first step towards creating an accurate disassembly for this application, which then would enable a human or static analysis tool to perform a security analysis of the code.

However, the ARM processor introduced a challenge: modern ARM processors support two related instruction sets. The ARM instruction set is a RISC-based set which contains 32-bit fixed-width instructions. An additional 16-bit fixed-width instruction set, called THUMB, is also supported. An application has the ability to link together both ARM and THUMB code and indicate to the processor which instruction set to execute. The division between THUMB and ARM instructions must be inferred by following control flow or by manual verification of code at a particular address.

An initial attempt was made to locate the absolute virtual address of the application segment located at the beginning of the firmware image. The virtual address for this segment was determined by locating constructs similar to C-language switch statements in compiled code. When switch case values are sequential and dense, the values are often converted into an index into a jump table that contains data values specifying the absolute virtual addresses of all the valid switch cases. This table allows the person disassembling the code to infer the absolute virtual address of a piece of nearby code.

For example, the code and data in Figure 3 was found within an initial disassembly of the firmware. In this case, the compiler has a switch construct with valid cases from 0 to 19. A test for invalid switch cases is performed at 0x12C64, with the BHI instruction at 0x12C66 branching to the default case. Following this, the switch case value is multiplied by 4 (via a left shift by two bits) and is used as an index into an array of 32-bit values that specify addresses for all valid known switch cases. In this example, it was possible to infer that the default case would be located at the virtual address 0x412D2E in memory. Since this code was at a file offset of 0x12D2E, we were able to infer that the code at the beginning of the file was mapped at 0x400000 in memory.

Upon making this assumption, we quickly noticed that several relative linked branches (an ARM equivalent to function calls) from within this executable segment referenced invalid locations at lower addresses. After navigating further into the firmware, we found that linked branches referenced invalid addresses at higher addresses, rather than lower addresses as was previously true for other portions of the code. By locating the split in the application where invalid linked branches stopped referencing lower addresses and began referencing higher addresses, we were able to locate a split in the memory mapping of the file. This allowed us to divide the file up into two distinct memory segments. We then

ROM:12C64	CMP	R2, #19] Bounds check and dispatch operation	
ROM:12C66	BHI	loc_12D2E		
ROM:12C68	LSL	R2, R2, #2		
ROM:12C6A	ADR	R3, dword_12C70		
ROM:12C6C	LDR	R2, [R3,R2]		
ROM:12C6E	MOV	PC, R2		

ROM:12C70	dword_12C70	DCD 0x412D24] List of switch case targets (dispatch table)	
ROM:12C74		DCD 0x412D1E		
...		...		
ROM:12CB4		DCD 0x412D2E		
ROM:12CB8		DCD 0x412D2E		
ROM:12CBC		DCD 0x412CC0		

ROM:12CC0	BL	sub_1435C] Case statement implementations (dispatch targets)	
ROM:12CC4	POP	PC		

...		...		
ROM:12D24	BL	sub_13BC4		
ROM:12D28	POP	PC		

ROM:12D2A	BL	sub_13684]	
ROM:12D2E	POP	PC		

Fig. 3. Switch code with absolute addressing given in the dispatch table. An attacker or analyst can use these addresses to identify the memory address at which the code would be loaded.

examined code beyond the split and looked for C switch statement constructs as was done before. With this information, we were able to create a file that had the correct data mapped at the correct virtual addresses, and at this point the disassembly began to become coherent.

The overall process to obtain a valid and accurate disassembly was long and tedious. However, there likely will not be a necessity to replicate this work for any future security analysis done on this platform because knowledge gained in this process could be automated and applied widely in the future. While the initial difficulty of creating a disassembly was a challenging hurdle to overcome, we do not believe this process is beyond the scope of what can be feasibly attained by security researchers or skilled attackers. As such, using an unknown but generally unobfuscated firmware format is not of real security benefit and does not deter security analysis of an embedded platform.

5.2 Binary Analysis

As with any binary security analysis of a large application or operating system, we focused our efforts on portions of the IP phone's firmware that were likely to

contain serious security vulnerabilities. This was done by using several commonly used manual auditing methodologies. As of the time of research and publication, we were not aware of any publicly available tools that would help perform an automatic security analysis of a compiled and linked ARM binary, so we worked manually in IDA Pro.

Although the Cisco IP phone runs an embedded operating system, the applications within this framework are developed in what appears to be the C programming language, and the developers have re-implemented the functionality present in most standard C runtime libraries. A primitive methodology used in our binary analysis was to identify functions within the disassembly that mapped to known C-runtime library functions. These could be identified by calling conventions, examination of function arguments, and the application's use of return values. Identifying string manipulation and dynamic memory allocation functions such as `strcpy`, `strcat`, `sprintf`, `strncpy`, `malloc`, `realloc`, and similar functions gave us a starting point for security analysis.

After these functions were identified, it was possible to follow their use within an application in order to look for potential misuse. A challenge with this approach is that the person performing manual binary review must have an idea of the data flow leading up to the potentially dangerous API call in order to properly assess the actual presence or severity of a potential vulnerability. Several potential vulnerabilities were identified through this methodology, however their true risk would not be fully known without a better understanding of the control and data flow through the applications on the IP phone. As such, a more formal analysis was undertaken to understand this aspect of the IP phone.

In order to understand more about the IP phone, we performed a functionality assessment to identify entry points for untrusted and attacker-supplied data. Our assessment of the IP phone was aided by the presence of debugging functionality in the firmware for the IP phone. By enabling a telnet server on the device, it was possible to obtain debugging output for many of the features of the IP phone. In many cases, this debugging output could be matched to the input that the IP phone received from an external source.

In addition to debugging output available on the console, there was a significant presence of debugging output within the firmware disassembly. Many of the critical functions within the firmware contained string references which indicated their likely original name within the source code or their intended functionality and purpose. The presence of function names within strings in the binary helped tremendously in the enumeration of likely input points for untrusted data.

After input points for untrusted data were identified, it was possible to enumerate possible sources of untrusted input and then begin a more in-depth security analysis. Some of the sources of untrusted input identified in our analysis were: Cisco Discover Protocol (CDP), Ethernet (802.3), IPv4, TCP, UDP, NTP, SIP requests, SIP responses, RTP, DNS responses, DHCP responses, TFTP responses, TFTP file downloads, SDP, media codec frames (g711ulaw, g729a), and telnet.

Our security analysis followed control and data flow from known untrusted input sources, looking for security vulnerabilities. We paid careful attention to common mistakes such as unbounded memory copy operations, loop constructs, integer issues, information leakage, and many other classes of vulnerabilities.

5.3 Results and Recommendations

The security quality of Cisco IP phone's SIP firmware initially appeared to be strong. Fuzz testing with publicly available testing suites for SIP did not lead to any instability in the device. However, manual security analysis revealed a more accurate assessment of the security quality of the code. Through manual analysis, it was possible to detect security flaws that would be very difficult to anticipate or test for through fuzz testing. Although there is evidence that the code has been reviewed for security, there were still serious vulnerabilities that were uncovered by manual security review.

In contrast to the runtime analyses of Section 4, our manual security analysis revealed weaknesses in the security of the IP phone's software. We identified both general architectural mistakes in the design of the software as well as implementation errors that can directly lead to successful exploits. Taken alone, the design flaws may not be security weaknesses. However, when combined with the other software vulnerabilities, they can cause simple attacks to rapidly escalate into full device compromise.

The fundamental design weaknesses center on the lack of memory protection among processes and the operating system itself. Most ARM processors allow for the separation of user-mode virtual system memory from the supervisor-addressable memory that is typically reserved for the operating system kernel. The Cisco 7960G IP phone appeared to have *no separation of privilege levels associated with its memory management model*. As such, a compromise in any portion of the IP phone can lead to full compromise of the device. Artificial security constraints, such as an unprivileged `telnet_level mode`, have very little meaning. An unprivileged telnet user who can cause memory corruption in any portion of the application can escalate her privileges to those of a fully privileged user.

Another potential security weakness became evident when examining debugging output from the "show stacks" command on the telnet terminal (Figure 4). After rebooting the IP phone several times, it was apparent that the stack addresses for each task on the IP phone were fixed. The predictable memory layout makes it extremely easy for an attacker to predict the exact location where data she has supplied might reside on the stack, increasing the reliability of memory corruption exploitation against these devices.

It is also evident that the stacks for different tasks are immediately adjacent to each other without any unmapped page or guard page in between. As such, memory corruption from one task's stack can directly influence the operation or control flow of another task. In a similar way, any recursive call patterns within the firmware code can lead to unintended corruption of the stack of an adjacent task. The stack grows down on all ARM processors we have experience working

```

SIP Phone> show stacks
Task: SOC (27) stkhi=0041b6ac stklo=0041beab Size=2048 Unused=1328
Task: RTP (26) stkhi=0041beac stklo=0041c6ab Size=2048 Unused=1916
Task: DSP (25) stkhi=0041c6ac stklo=0041ceab Size=2048 Unused=1732
Task: PHN (24) stkhi=0041ceac stklo=0041deab Size=4096 Unused=2108
Task: GSM (23) stkhi=0041deac stklo=0041f6ab Size=6144 Unused=5972
Task: SIP (22) stkhi=0041f6ac stklo=00421eab Size=10240 Unused=7476
Task: GUI (21) stkhi=00421eac stklo=00422eab Size=4096 Unused=1012
Task: NET (19) stkhi=00422eac stklo=004236ab Size=2048 Unused=624
Task: CFG (18) stkhi=004236ac stklo=00423eab Size=2048 Unused=1052
Task: TTY (17) stkhi=00423eac stklo=00424eab Size=4096 Unused=3280
Task: AUD (16) stkhi=00424eac stklo=004256ab Size=2048 Unused=1688
Task: PTMR (29) stkhi=004256ac stklo=00425eab Size=2048 Unused=1924
Task: TMR (28) stkhi=00425eac stklo=004266ab Size=2048 Unused=1148

```

Fig. 4. Output of the `show stacks` telnet command.

with. On this architecture, it appears that recursive control flow, a relatively common coding practice, can lead to compromise of the device.

Implementation flaws can provide the entry point for an attacker to leverage the architectural weaknesses. Our binary analysis uncovered four coding errors: *one heap-based buffer overflow* and *three stack-based overflows*. In this public document, we will present only a high-level overview of the vulnerabilities.

First, we analyzed the heap overflow. In non-embedded systems, heap overflows are often exploited by corruption of heap control structures that are stored in-band with application data, or by direct manipulation of application-specific data on the heap. Generally speaking, heap implementations that store heap control structures out-of-band of application data are less susceptible to exploitation than those that store critical information in a place easily corrupted by application vulnerabilities.

We performed a cursory examination of the Cisco IP phone's heap implementation to assess the severity of heap corruption on the platform. Although there are heap control structures in-band on the heap, there appears to be sanity checking of at least certain portions of the heap structure, including the field specifying the heap chunk size. Corruption of application data prior to freeing of the corrupt block seems to be a viable vector for exploitation. There is a good possibility that heap corruption can lead to reliable remote code execution, although it might be less straightforward than on many non-embedded systems.

Figure 5 shows the assembly code of this vulnerability. We have redacted the full assembly code addresses to prevent straightforward discovery of the vulnerable code by unskilled attackers. The attack can occur if when a specially-crafted message is sent to the phone in response to a SIP request sent by the phone to its proxy. When the IP phone receives the malicious message, register R5 points to a pre-parsed list of authentication parameters from the message,

```

ROM:XXXXXX36      MOV R6,#0
ROM:XXXXXX38      CMP R5,R6
ROM:XXXXXX3A      BEQ loc_XXX4A
ROM:XXXXXX3C      MOVL R0,0x400
ROM:XXXXXX40      BL malloc
ROM:XXXXXX44      ADD R4,R0,#0
ROM:XXXXXX46      CMP R4,R6
ROM:XXXXXX48      BNE loc_XXX4E
...
ROM:XXXXXX4E      LDR R0,[R5,#4]
ROM:XXXXXX50      CMP R0,#2
ROM:XXXXXX52      BEQ loc_XXX62
...
ROM:XXXXXX62      LDR R0,[#0x10]
ROM:XXXXXX64      STR R0,[SP,#0x38+var_38]
ROM:XXXXXX66      ADR R0,aRealm ; realm
ROM:XXXXXX68      STR R0,[SP,#0x38+var_34]
ROM:XXXXXX6A      LDR R0,[R5,#0xC]
ROM:XXXXXX6C      STR R0,[SP,#0x38+var_30]
ROM:XXXXXX6E      ADR R0,aUri_0 ; uri
...
ROM:XXXXXX8A      ADR R2, aDigest ; Digest
ROM:XXXXXX8C      ADR R3, aUsername ; username
ROM:XXXXXX8E      BL sprintf_0

```

Fig. 5. Assembly code showing the SIP authentication request vulnerability in the Cisco 7960G.

such as realm, URI, and nonce. The IP phone allocates a statically-size heap buffer of 0x400 (1024) bytes at ROM:XXXXXX40. The software then performs an unbounded `sprintf` function call into this heap buffer at ROM:XXXXXX8E. If the sum of the string lengths and count of characters present in the format string is greater than 1024, then a heap overflow occurs. While this vulnerability is easy to exploit, it requires special preparation on the part of the attacker, making it difficult to test automatically. The attacker must first set up a man-in-the-middle attack and then wait for the IP phone to send a request to its proxy.

The remaining three vulnerabilities discovered in this security review were stack corruption issues. We found two of the stack corruption vulnerabilities in portions of the code that could not be easily tested by fuzz testing with random input. In one case, a stack corruption vulnerability required authentication and was therefore limited to only a privilege escalation issue.

In another case, a stack corruption vulnerability was remotely exploitable by sending a crafted SIP authentication request message to the IP phone. To exploit this vulnerability, an attacker waits for the IP phone to send a SIP request (i.e., INVITE) to its proxy and then, using a man-in-the-middle attack,

the attacker sends back to the IP phone a malicious *407 Proxy Authorization Required* message. A stack overflow is produced by the malicious message when the IP phone tries to parse it to compute the response to the authentication request.

A final stack corruption issue uncovered by our analysis has no known mitigating factors preventing exploitation, and is likely exploitable in virtually every functioning configuration of the device. An attacker can exploit this vulnerability with a single network packet (a SIP INVITE message). This final error is likely the most serious of the issues discovered in this security review, and the only viable solution for this flaw in many organizations will be to deploy intelligent network filtering or to upgrade the firmware for all IP phones to a patched version. This can be challenging for large enterprises to deploy efficiently, and patches will likely have to be created for many different IP phone protocol firmware versions.

In order to better understand the security consequences of stack corruption on this platform, we examined the firmware for additional security measures that might make exploitation more difficult. Surprisingly, there were no security features present to detect or limit the impact of stack corruption. Although some of the most effective stack protection technologies might not be feasible on an embedded platform due to processing and resource constraints, there are some simple protection features that are widely used and have minimal memory and computational footprints. For example, compile-time additions such as Stack Guard [17] can detect many instances of stack corruption at runtime via intelligent stack frame reordering and the placement of a randomly-seeded canary value adjacent to critical register values stored on the stack. The addition of this type of feature to a Cisco IP phone would likely require little additional effort on the part of developers and would help prevent exploitation of basic stack corruption vulnerabilities. At the moment, there is nothing to prevent an attacker who is able to corrupt the stack from redirecting execution to an arbitrary location in memory and executing arbitrary code on a Cisco IP phone.

Exploiting memory corruption vulnerabilities on an ARM architecture can be challenging because the processor uses split data and instruction caches. The data cache operates in a write-back mode in which writes from cache to physical memory are delayed until the dirty cache line is evicted. This lack of coherency between the data and instruction caches is a challenge for exploits of memory corruption vulnerabilities. In a typical memory corruption exploit, an attacker will supply shellcode as part of the attack. This shellcode is comprised of assembly instructions that will eventually be executed by the vulnerable application if the attack is successful. This code is written to memory by the application or as a result of network I/O prior to it being executed.

However, in the case of an ARM processor, the writing of shellcode to memory is delayed by the data cache until it is drained and the changes are flushed out of cache back into main memory. If this does not occur prior to the attacker redirecting control to the location where the code is supposed to reside, then the instruction cache will fetch instructions from main memory prior to them being

flushed out of the write buffer or data cache. As such, the instructions that will end up being executed are not those supplied by the attacker and the attack will generally fail. It is possible to drain write buffers and flush data cache by making use of the coprocessor register 15 (cp15) in a specific manner on most modern ARM processors. It is also possible to cause the cache to get flushed by inducing certain memory accesses. This might be possible by causing a Read After Write (RAW) in some cases, and this type of cache manipulation has been proven successful in publicly documented vulnerability research projects [18]. Further investigation will likely yield the most efficient way to achieve this on a Cisco IP phone.

6 Related Work

In the area of VoIP vulnerability analysis, Abdelnur et al. [19] proposed a network information model able to represent the information required to perform VoIP assessment and a framework based on attack tree modeling in order to represent and write VoIP attacks. In other work, Abdelnur et al. [12] designed and implemented a stateful protocol fuzzer for SIP capable of tracking the state of the targeted application and device. This tool was used to test a Cisco IP Phone 7940, part of the same family and firmware as the 7960G, and was able to find vulnerabilities different from those we found using static binary analysis. McGann and Sicker [20] presented a study of the known VoIP-related vulnerabilities and the results of testing several of the more popular open source and commercial VoIP security tools. Their results showed that many of these security tools do not cover the extend of the known vulnerabilities. We reached a similar conclusion from our study.

Binary analysis techniques have also been applied to other embedded devices. Fogie [18], San [21], Hurman [22], and Mulliner [23] demonstrated the feasibility of reverse engineering, memory corruption attacks, and shellcode development to exploit vulnerabilities in mobile devices running Windows CE operating system on ARM processors. Similar work has been done in the area of networked embedded systems, where FX [24, 25] and Lynn [26] described several vulnerabilities and exploits against Cisco network devices. Barnaby [27] showed how to reverse firmware using a JTAG debugging interface. Finally, Grand [28] and O'Connor [29] presented several techniques to compromise other embedded systems such as mobile phones, printers, scanners, and similar devices.

Security mechanisms to protect embedded systems have been also published by several academic and industry researchers. Verma [30] described a Texas Instruments technology, named *Static Packet Filter* (SPF), which consisted of a programmable engine that can be configured into an IP phone to protect it against many different types of DoS attacks. Shao et al. [31] developed a protection and checking mechanism to protect embedded systems against buffer overflow attacks and efficiently check if a component has been protected, even without the presence of source code. Arora et al. [32] described a mechanism for secure program execution in embedded systems based on a hardware monitor

that can be connected to any embedded processor to observe its dynamic execution trace as it executes the program, check whether the trace conforms to the definition of permissible behavior, and flag violations by triggering appropriate response mechanisms.

7 Conclusions

The evolution from traditional telephony to VoIP systems is adding more intelligence, functionality, and complexity to the end points. This increasing complexity is making the end points vulnerable to many of the security problems that affect today's desktop computer systems. In the same way, security analysis techniques commonly used in desktop computer systems, such as binary analysis, can also be used to find vulnerabilities in embedded devices like IP phones, as we showed in this paper.

Even though the IP phone was resilient to all the vulnerability scans and fuzzing attacks that we attempted, the disassembly and static analysis was able to uncover serious flaws: a design weakness and four memory corruption vulnerabilities that pose significant risks to organizations deploying the IP phone. We can conclude from this experience that while automated tools can help to detect security problems more efficiently, it can be risky to rely only on that kind of tool to assess the security of these devices. Automated security tools have to continue improving their techniques to detect new vulnerabilities in embedded systems such as IP phones, and new tools have to be developed to automate techniques such as static binary analysis in embedded platforms.

While we specifically selected the Cisco IP phone due to its relevance and the lack of comprehensive security research on the subject, there are broader implications that can be drawn from this research. It is safe to assume that similar vulnerabilities exist for a great deal of other IP phones in particular and embedded devices in general, and their increased deployment in a variety of uses poses a significant security risk. The historical pattern of coding errors from major software developers indicates that security vulnerabilities will not disappear without significant effort on the part of the development and security communities. Until better secure development practices and sufficient security testing are performed before embedded devices such as Cisco IP phones are released into the marketplace, there is no reason to assume that these vulnerabilities will be eradicated.

Acknowledgments. This work was sponsored in part by AT&T and IBM Internet Security Systems.

We thank the anonymous reviewers and the VoIP research group at Georgia Tech for their helpful feedback that improved the quality of this paper. We thank IBM ISS for generously providing test equipment used in this study.

References

1. Bell, A.G.: Improvement in telegraphy. United States Patent #174,465 (March 1876)
2. Prevelakis, V., Spinellis, D.: The Athens affair. *IEEE Spectrum* **44**(7) (July 2007)
3. Arkin, O.: The trivial Cisco IP phones compromise. Whitepaper, The Sys-Security Group (September 2002)
4. Cisco Security Advisory: Cisco unified IP phone overflow and denial of service vulnerabilities. <http://www.cisco.com/warp/public/707/cisco-sa-20080213-phone.shtml> (2008)
5. Cisco Press Release: Cisco sells its 6 millionth IP phone as worldwide demand soars for IP communications (September 2005)
6. Kuhn, D., Walsh, T., Fries, S.: Security Considerations for Voice Over IP Systems. US Dept. of Commerce, National Institute of Standards and Technology (2005)
7. Hoglund, G., McGraw, G.: *Exploiting Software: How to Break Code*. Addison-Wesley Professional (2004)
8. Raghunathan, A., Ravi, S., Hattangady, S., Quisquater, J.J.: Securing mobile appliances: new challenges for the system designer. In: Design, Automation and Test in Europe, Munich, Germany (March 2003)
9. Kocher, P., Lee, R., McGraw, G., Raghunathan, A., Ravi, S.: Security as a new dimension in embedded system design. In: Design Automation Conference, San Diego, CA (June 2004)
10. Secunia: Cisco IP phone 7960—vulnerability report. <http://secunia.com/product/287/?task=advisories> (2007)
11. VoIPSA: Voip security tool list. <http://www.voipsa.org/Resources/tools.php> (2007)
12. Abdelnur, H., State, R., Festor, O.: KiF: A stateful SIP fuzzer. In: 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm), New York, NY (July 2007)
13. Cisco SIP IP Administrator Guide, Version 8.0. http://www.cisco.com/en/US/docs/voice_ip_comm/cuipph/7960g_7940g/sip/8-0/english/administration/guide/8.0.html (2007)
14. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: RFC3261: SIP: Session initiation protocol (2002)
15. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. *Communications of the ACM* **33**(12) (December 1990)
16. State, R.: Cisco phone 7940 remote DOS. CVE-2007-5583 (2007)
17. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium, San Antonio, TX (January 1998)
18. Fogie, S.: Embedded reverse engineering: Cracking mobile binaries. In: Defcon 11, Las Vegas, NV (2003)
19. Abdelnur, H., State, R., Chrisment, I., Popi, C.: Assessing the security of VoIP services. In: 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany (May 2007)
20. McGann, S., Sicker, D.: An analysis of security threats and tools in SIP-based VoIP systems. In: 2nd Workshop on Securing Voice over IP, Cyber Security Alliance, Washington, DC (June 2005)
21. San: Hacking Windows CE. In: Defcon 13, Las Vegas, NV (2005)

22. Hurman, T.: Exploring Windows CE shellcode. Whitepaper, Pentest Limited (June 2005)
23. Mulliner, C.: Advanced attacks against PocketPC phones. In: Defcon 14, Las Vegas, NV (2006)
24. FX: Attacking networked embedded systems. In: Black Hat Windows Security, Seattle, WA (February 2003)
25. FX: More embedded systems. In: Black Hat USA, Las Vegas, NV (July 2003)
26. Lynn, M.: The holy grail: Cisco IOS shellcode and exploitation techniques. In: Black Hat USA, Las Vegas, NV (July 2005)
27. Barnaby, J.: Exploiting embedded systems. In: Black Hat Europe, Amsterdam, Netherlands (Feb/Mar 2006)
28. Grand, J.: Introduction to embedded security. In: Black Hat USA, Las Vegas, NV (July 2004)
29. O'Connor, B.: Vulnerabilities in not-so embedded systems. In: Black Hat USA, Las Vegas, NV (Jul./Aug. 2006)
30. Verma, A.: IP phone security: Packet filtering protection against attacks. Texas Instruments White Paper (2006)
31. Shao, Z., Xue, C., Zhuge, Q., Qiu, M., Xiao, B., Sha, E.H.M.: Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. *IEEE Transactions on Computers* **55**(4) (April 2006)
32. Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Hardware-assisted run-time monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **14**(12) (December 2006)