Hacking Video Conferencing Systems

Moritz Jodeit (n.runs AG)
moritz.jodeit@nruns.com

Abstract

High-end videoconferencing systems are widely deployed at critical locations such as corporate meeting rooms or boardrooms. Many of these systems are reachable from the Internet or via the telephone network while in many cases the security considerations are limited to the secure deployment and configuration.

We conducted a case study on Polycom HDX devices in order to assess the current state of security on those devices. After analyzing the software update file format and showing how to get system level access to the otherwise closed devices we describe how to setup a proper vulnerability development environment which lays the groundwork for future security research.

We demonstrate the feasibility of remotely compromising Polycom HDX devices over the network by implementing an exploit for one of the vulnerabilities we identified in the H.323 stack of the current software version which allows us to compromise even firewalled devices as long as the H.323 port is reachable. Our attack does not require the auto-answer feature for incoming calls to be turned on.

We conclude with some thoughts about post-exploitation and describe possible ways to control attached peripherals such as the video camera and microphone which could be used to build a surveillance rootkit.

1 Introduction

Nowadays videoconferencing systems are widely deployed in the business, government, medicine and education sectors. Companies are using those systems to connect executive level meeting rooms which are geographically spread around the globe. Governments make heavy use of video conferencing for conducting meetings between political leaders [8]. But it is also used for initial court appearance or to carry out hearings at remote locations [12]. The military is using videoconference systems for information exchange in remote tactical locations but also for soldiers to communicate with their families. Finally videoconferencing systems are also used for training purposes in the education sector as well as for telemedicine applications in the healthcare industry.

In many of the mentioned scenarios classified information is transmitted over those systems. Even if the videoconference system is only used for non-classified information, devices might be installed at important locations such as boardrooms where the most critical meetings take place. With the equipped HD camera and highly sensitive omnidirectional microphones modern systems provide, those devices could easily be used to listen in on conversations or read sensitive information lying on a conference room table, opening the door for industrial espionage. Additionally videoconferencing systems are often accessible over public networks such as the Internet or the telephone network. Exploiting those devices would not only allow an attacker to eavesdrop on communication, but could also provide a first foothold for intrusions into the internal company network.

In this paper we present the results of our security analysis of a Polycom HDX system and demonstrate the feasibility of remotely compromising those devices.

In the following sections we describe in detail the format of the firmware update files and show how to unpack them in order to conduct a proper security analysis of the software.

Based on these results we show different ways to get system level access to the otherwise closed devices and propose a possible solution for rooting the device in the future even after all the described vulnerabilities have been fixed by the vendor.

We describe the architecture of the system and the main system processes which should be audited for potential security vulnerabilities. We show how to setup remote debugging to allow a proper vulnerability development environment to be created.

Based on one of the vulnerabilities we identified in the H.323 stack in the latest software version we describe a way to remotely exploit a Polycom HDX system in a firewalled environment where all management interfaces are disabled or firewalled off.

We conclude our analysis with a discussion about payload development which can be used in a post-exploitation scenario to control connected peripherals such as the video camera or the microphone which could be used to implement a surveillance rootkit.

In the next section we give some background information about videoconferencing in general.

2 Background

Videoconferencing systems allow two or more parties to visually communicate and collaborate. Video, audio and optionally data are transmitted over a digital packet-based network such as IP or ISDN, allowing face-to-face communication even if the meeting members are located on different continents.

There are basically two different kinds of videoconferencing systems: Dedicated systems and Desktop systems. Dedicated systems are appliances which come with all the necessary equipment. They typically come with a pan-tilt-zoom (PTZ) video camera which can be remotely controlled. Additionally several omnidirectional microphones are provided. The console is an embedded system which connects all the components together and provides the control interface and the software or hardware codecs. Dedicated systems are typically connected to a television set or projector.

Dedicated systems come in different sizes. They range from large, non-portable systems (e.g. dedicated for fixed installations at large conference rooms) down to small portable devices which provide the video camera, microphone and speakers all in the console for mobility.

Desktop systems on the other hand use an existing workstation and make use of the connected web cam, microphone and speakers which makes them less expensive. Desktop systems either come as hardware extensions or as software add-ons.

3 Polycom HDX Devices

Polycom is the leading vendor for unified communications (UC) systems. They offer different dedicated systems for telepresence, video, and voice. The most popular videoconferencing units can cost up to $25,000. Due to the high price these devices are mostly used by larger companies and organizations. According to Polycom their devices are used by leading government agencies and ministries worldwide as well as by the world's 10 largest banks, and the 6 largest insurance companies worldwide [9].

We are targeting the HDX series of devices which is a popular videoconferencing solution sold by Polycom. All tests described in this paper were conducted on a HDX 7000 HD device.

The HDX 7000 HD device is an embedded system with a PowerPC e300c1 CPU running Linux. It comes with 256 MB RAM, a 512 MB CompactFlash disk to store the operating system and onboard flash memory to store the boot loader. Our HDX device was equipped with the external Polycom EagleEye HD camera which is a PTZ video camera but also serves as the infrared (IR) receiver for the HDX IR remote control. The system also came with the Polycom HDX Mica Microphone Array which is a table microphone for 360-degree audio pickup.

3.1 Attack Surface

This section lists some of the network-based interfaces of the system. It is meant to give the reader an idea of the possible attack surface. However it's not meant to be exhaustive. For a complete list of interfaces and supported protocols see the administrator's guide at [10].

HDX systems offer several different administrative interfaces to remotely access the system. The main administrative interface is the Polycom HDX Web Interface. The web interface is the recommended way to configure all aspects of the HDX system. It also offers diagnostics and remote control functionality.

Instead of manually configuring the system through the web interface it is also possible to use a provisioning service. With the help of the Polycom Converged Management Application (CMA) or the RealPresence Resource Manager system it is possible to centrally manage HDX devices. This way it is possible to deploy configuration settings and software updates remotely.

Another administrative interface provided by the HDX system is the Application Programming Interface (API). The API is a set of commands to automate a Polycom HDX system. This interface is meant to be used by advanced users or system integrators. The API can be accessed over the RS-232 serial console or over the network via telnet on port 24. For a complete list of possible commands see the integrator's reference manual at [11].

HDX devices provide another console-based administrative interface, which is called the Polycom Command Shell. It can be accessed via telnet on port 23 and can be used to remotely control the device. It offers commands to view and change various device settings and can also be used to control the device and attached components such as the video camera. A list of supported commands can be printed with the `help` command.

Finally HDX devices support SNMP for monitoring the system for certain events such as failed and successful admin logins, connects and disconnects of phone or video calls, and various other events. Even though HDX devices only allow read-only access via SNMP. Write operations are not supported.

In addition to the administrative interface HDX devices implement the usual videoconferencing protocols such as H.323 and SIP.

3.2 Firmware Analysis

Software updates for all Polycom devices can be found at the Polycom Support page at http://support.polycom.com. This page provides software updates as well as documentation on the full range of products. The updates for the Polycom HDX series of devices come as ZIP files which contain a single PUP file which is the actual firmware update file. The current version as of this writing is 3.0.5-22695. All the following analysis described in this paper is based on this version.

Software updates can either be installed manually via the web interface or can be installed automatically when a provisioning service is used. In the latter case the system will check for new updates every time it restarts and at an interval set by the system.

In order to analyze the Polycom update (PUP) file format, we first scanned the file for known signatures using Binwalk [3]. It turns out that the first 768 bytes of the file represent the PUP header. The header is followed by a gzip compressed `tar` archive and an additional uncompressed `tar` archive. The two archives are separated by the string constant "`--multipart boundary 1--`". The first `tar` archive contains the bootstrap code which is used to install the update. The main functionality can be found in the embedded `setup.sh` shell script. The second `tar` archive contains the actual update. All files embedded in this archive are extracted and copied to the HDX system, potentially overwriting older files. A simple representation of the PUP file structure is shown in Figure 1.

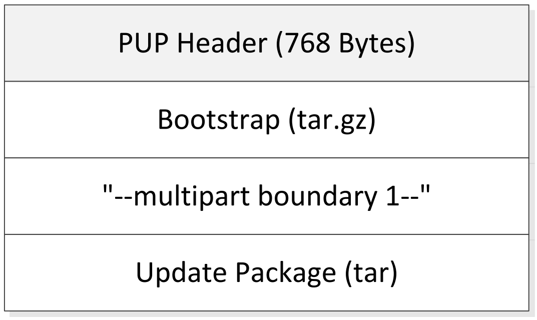


Figure 1: PUP file format

Instead of manually trying to figure out the format of the PUP header, we are going a different route. Looking through the embedded update archive, we found the binary `polycom.g3/polycom/bin/puputils.ppc`. This is the PUP Utility which is used on the HDX system to verify and install new firmware updates. Additionally it also provides the functionality to create new PUP files.

By reverse engineering this binary we were able to figure out the exact PUP header format without any guess work. The complete format is shown in Table 1.

The PUP header consists of several fixed-size fields

length (bytes)	**description**
5	PUP File ID
4	Header Version
20	Header MAC Signature
32	Processor Type
32	Project Code Name
16	Software Version
16	Type of Software
32	Hardware Model
16	Build Number
32	Build Date
16	Build By
16	File Size (without header)
5	Compression Algorithm
445	Supported Hardware
81	Signature (ASN.1 encoded)

Table 1: PUP header file format

which are padded with null bytes. Every PUP file starts with the fixed PUP file ID "PPUP" or "PPDP" followed by a single null byte. The file ID is followed by the header version after which comes a 20 byte cryptographic MAC value which will be described shortly.

Most of the following fields are pretty self-explanatory and describe either properties of the firmware update (such as version or build date) or requirements of the target hardware. The last field of the header contains a public key signature which is stored in ASN.1 encoded form.

The first protection against file tampering is the embedded Header MAC Signature which purpose seems to be to protect the PUP header from being modified. When the firmware update is installed the message authentication code (MAC) of the PUP header will be calculated (setting the Header MAC Signature field to zero). The result will be compared against the MAC stored in the PUP header. If they don't match, the firmware update process will be aborted.

The obvious problem with using a MAC for integrity checks is that the secret key used to calculate and verify the MAC must be stored on the device. In this case it is stored inside the `puputils.ppc` binary itself. The secret key consists of the string "`weAREtheCHAMPIONS`" prefixed and followed by five constant seemingly random bytes respectively.

We didn't spend any further time reverse engineering the used MAC algorithm. Modifications to the PUP header can easily be done by first running the extracted `puputils.ppc` inside a virtual machine on the modified PUP file. We used a Debian Squeeze image running inside Qemu to emulate the PowerPC environment which allowed us to execute the binary without a problem. It's not even necessary to use a debugger to extract the correct MAC value calculated by the tool, because the

PUP Utility will happily print the expected and valid MAC for the given file in the error message as can be seen in Listing 1.

We just have to replace the MAC inside the PUP header with the printed value to get a PUP file with a valid MAC signature.

In addition to the PUP header MAC a public key DSA signature is used to verify the integrity of the whole file (including the PUP header). The signature is stored at the end of the PUP header. When `puputils.ppc` is executed to verify a PUP file it first reads the DSA public key file `su_public.key` from the current working directory and then uses it to verify the signature stored in the PUP file. If either the PUP header or the body of the file were modified the signature verification fails. So it is not enough to just break the PUP header MAC to install modified software updates.

We didn't spent any further time analyzing the signature verification process in detail. However for rooting the device there are easier methods to accomplish this goal which are described in the next section.

3.3 Rooting the Device

Polycom HDX systems are closed systems, i.e. the customer is not supposed to get direct access to the underlying system. Although we can do bug hunting without any system level access, it is much more comfortable to have control over the device. Especially when it comes to fuzzing, monitoring running processes for faults and controlling the device (e.g. restarting processes) is important. Having root access also simplifies the exploit development process a lot. In this section we are taking a look at several different approaches on how to get root access to the device.

3.3.1 Polycom Command Shell

There are several possibilities to get root access to the system. The easiest way however is through the Polycom Command Shell. This shell can be accessed either via a RS-232 serial connection or via telnet on port 23. By analyzing the extracted firmware update we found out that the Polycom HDX system can be booted in two different modes. By default it is running in Production mode. However the system can also be started in Development mode. The development mode turns the flash-based system to use an NFS-mounted developer workspace. Additionally it enables telnet access on port 23 where you can login with the user `root` without a password. This mode is used by Polycom internally for software development. However it can still be enabled in the released firmware.

To enable the development mode, we make use of the `printenv` and `setenv` commands in the Polycom Command Shell. The `printenv` command allows us to display all the configuration variables which are stored in flash memory. With the `setenv` command we can modify those. The startup script `/etc/init.d/rcS` checks for the U-Boot environment variable `devboot` and boots into development mode if it is set. In order to set this variable, we can add the `othbootargs` variable with the value "`devboot=bogus`" in flash memory. The value of this variable will be appended as an additional kernel parameter and consequently enable the development mode on the next reboot. After the system booted in development mode, you can access it by connecting via telnet and login as the user `root` without a password. Listing 2 shows how to enable the development mode via a serial connection.

In development mode not all services are running. End-user services like the web interface are not started which makes this mode unsuitable for a bug hunting environment. However we can switch back to the production mode after adding a permanent way for us to access the device. We just enable a telnet server in the `/etc/inetd.conf.production` file on an unused port. This is the `inetd` configuration file used in production mode. To leave the production mode, you can either manually remove the `othbootargs` variable again using the `fw_setenv` command or call the `/opt/polycom/bin/devconvert` shell script with the command line argument "`normal`". This will reboot the system to re-enable the production mode.

3.3.2 Exploiting Command Injection

Another possibility to get system level access to the device is to exploit a vulnerability in one of the many interfaces provided by the HDX system. It is pretty straightforward to find command injection vulnerabilities in the current software version 3.0.5. We used a command injection vulnerability in the firmware update functionality itself which can be accessed through the web interface.

When uploading a PUP file via the web interface the file is first stored on the device and then the filename is passed as an argument to a call to the `puputils.ppc` binary in order to verify its integrity. Missing input validation allows us to inject additional shell commands by using shell metacharacters (such as a semicolon). We just renamed a valid PUP file to mount this attack. One limitation is that we can't use slash characters directly because the injected command is part of a path and only the filename (the part after the last slash character) is used in the call to the `puputils.ppc` binary. Another minor limitation is the fact that upper case characters are converted to lower case. Although we can easily workaround those limitations by using shell command substitution to generate the desired characters, we chose to only use the command

```
$ ./puputils.ppc verify modified.pup hdx
pc[0]: Welcome to the PUP Utilities.
pc[0]: Verifying the integrity of the PUP file "modified.pup"

pup file SHA-1 Hash: (160-bit)
11876296a8d432841de41526200543caf10ab020
pc[0]:  {1} Verified that we are working with a .pup file.
pc[0]: {2} PUP header version = 002

MAC: (160-bit)
5c3aa27774bd22ff98a1bd95aef09b3b1e11c6f0
pc[0]: The MAC does not match! The PUP header appears to have
       been tampered with.
pc[0]: returning PUP_ERR_HDR_MAC_MISMATCH
```

Listing 1: PUP utility prints expected MAC value of modified PUP file

```
$ cu -l ttyUSB0 -s 9600
-> setenv othbootargs "devboot=bogus"
-> reboot
reboot, are you sure? <y,n> y
```

Listing 2: Enabling development mode on a HDX system via a serial connection

injection to turn the system into development mode from which it is much more comfortable to work from. To enable development mode we have to inject the command `/opt/polycom/bin/devconvert bogus`. We replace every forward slash with the shell command substitution `` `pwd|cut -c1` `` which generates the forward slash for us. Running this command typically enables development mode and finally reboots the system. However after injecting this command the system doesn't reboot. We suspect that it's related to the fact that the system is running in update mode when we trigger the exploit. However waiting a few minutes and then power cycling the system will successfully boot into development mode and we are able to login via telnet as `root`.

3.3.3 Firmware Downgrade

The problem with the described ways of rooting the device is that they are not long-lasting since the used vulnerabilities will most probably be fixed by Polycom in future software updates. One obvious solution would be to just find another vulnerability. However an easier possibility is to just downgrade the running software version to one which is known to be vulnerable. After rooting the device via a known vulnerability we just upgrade back to the current software version. Since we have full control over the device during the upgrade process, we can make sure to keep the root access even after the software upgrade was applied.

3.4 System Architecture

The HDX system is an embedded PowerPC-based Linux system running Linux kernel 2.6.33.3. It's using the U-Boot boot loader. It comes with most of the standard Linux binaries including `busybox`, `wget` and even a `gdbserver` binary. The main storage has four partitions. The first partition `/dev/hda1` contains boot-related files and the Linux kernel image. The root file system can be found in `/dev/hda2` and it's mounted read-only. The third partition `/dev/hda3` stores all the changing files such as log and configuration files. It's mounted on the `/data` directory. The fourth partition `/dev/hda4` hosts the factory restore filesystem which contains the restore image `restore_image.tgz`. This file contains the system software which is installed when a factory restore is performed. The partition is not mounted in production use.

All the Polycom-specific files can be found inside the `/opt/polycom/` directory. Polycom binaries reside in the `bin/` subdirectory while configuration settings are stored in `.dat` files in the `/opt/polycom/dat/` directory. Each `.dat` file stores a single configuration setting which consists of one or more lines of text.

3.4.1 System Startup

Once the kernel was loaded the `/etc/init.d/rcS` script is executed which brings up the main services of the HDX system. The system can either boot in production or development mode which is decided by the `devboot` U-Boot environment variable.

If the system boots in production mode, one of the first things it checks is if the USB Diag Mode should be enabled. In this mode a diagnostic console is launched on an attached USB serial device. To check if this mode should be enabled, the system looks for any attached USB storage devices and tries to locate the file `diags.pdp`. This file is called the Polycom Diagnostic Package. If that file is found on an attached USB storage device, the validity of it is first verified with the `puputils.ppc` binary. If validation succeeds, the file is extracted, an embedded shell script is executed and the USB diagnostic mode is enabled. For further details refer to the `/opt/polycom/bin/usb_diags` shell script.

If the system is booted in development mode it tries to mount a developer workspace via NFS. The address of the remote NFS server is stored inside the `devboot` variable. The value "bogus" can be used to skip the NFS mount. In addition to mounting an NFS share, a telnet server is spawned on port 23 which allows logins with the user `root` without a password.

3.4.2 Main Processes

The main functionality of the HDX system is provided by two different processes which communicate with each other. The first process is called Polycom AVC and it's implemented in the `/opt/polycom/bin/avc` binary. This is a huge non-stripped binary of around 50 MB implemented in C and amongst many other things it contains all the implementations for the videoconferencing protocols like H.323 and SIP.

The other main process is based on Java and its implementation is scattered around several JAR files. Among other things this process is responsible for the GUI and also implements most of the administrative web interface functionality.

Both processes are running as the user `root` and communicate via XCOM, which is Polycom's IPC mechanism. The XCOM server is provided by the Java process and implemented in the `polycom.nativeaccess.XCOMConnection` class. By default it listens on localhost on port 4121. The XCOM protocol itself is a simple text-based protocol which is described in Section 6 in more detail.

The web server which provides the administrative web interface is based on lighttpd. The basic functionality as well as authentication is implemented in the Java process. The communication with the Java components happens via FastCGI.

4 Bug Hunting

When starting to look for bugs the main processes mentioned in the previous section are a good starting point. If the goal is to find vulnerabilities in the administrative web interface, the JAR files implementing the main functionality of that interface can be decompiled and analyzed for potential bugs. The JAR files can be found in the `/opt/polycom/bin` directory. Every Java class handling CGI requests extends the `polycom.web.CGIHandler` class and thus can easily be identified in further code reviews.

If the final goal is to find vulnerabilities in one of the supported videoconferencing protocols, the `avc` binary should be reverse engineered. In the current firmware update this binary is not stripped which eases the reverse engineering process and makes it easy to find the relevant code. From a bug hunting perspective the binary looks pretty interesting as well. Just to give an example, there are more than 800 cross-references to the `strcpy` function and more than 1400 calls to the `sprintf` function. Obviously this by itself is no proof for vulnerabilities but is a good indicator for the general code quality. Additionally the binary is not compiled with any exploit mitigations, simplifying the exploitation of any potential vulnerability.

Although bug hunting can be done solely based on the extracted firmware files, it's always helpful to be able to debug the running software. Especially when it comes to exploit development, having a working debug environment can save some time.

4.1 Remote Debugging

Debugging directly on the HDX system is no real option because of memory constraints. However the system comes with a `gdbserver` binary which can be used. On the HDX system we just execute "`gdbserver :1234 --attach `pidof avc``" to attach to the running `avc` process.

On the debug client host we just need a version of GDB which was compiled for the `powerpc-linux` target architecture. It is important that we let GDB know where the remote shared libraries reside on our system. If this step is skipped GDB will be able to attach, but breakpoints won't work and kill the process. So after starting `powerpc-linux-gdb` from inside the unpacked firmware's `polycom_swupdate/` directory, we can set the `solib-absolute-prefix` and `solib-search-path` GDB variables to achieve the desired effect. Listing 3 shows how to remotely attach to the debug stub running on the HDX system.

4.2 Watchdog Daemon

The HDX system is using a watchdog daemon which automatically detects crashes and processes which are not responding and automatically reboots the system in such a case. This behavior is undesirable since it will reboot the HDX system once we stop the `avc` process inside the debugger for too long or trigger a crash through fuzz testing.

```
$ pwd
/firmware/polycom_swupdate
$ powerpc-linux-gdb polycom/bin/avc
[...]
(gdb) set solib-absolute-prefix nonexistent
(gdb) set solib-search-path ./lib:./usr/lib:./polycom/bin
(gdb) target remote 10.0.0.1:1234
Remote debugging using 10.0.0.1:1234
[...]
```

Listing 3: Remote debugging the `avc` process on the HDX system

To disable the watchdog daemon we can't just kill the `watchdogd` process, since this will be followed by a system reset. However by reverse engineering the `watchdogd` binary we found a way to disable the watchdog daemon with a configuration setting. When the daemon is started at system boot time it will check for the existence of the `/opt/polycom/dat/watchdog_disable.dat` file. If that file exists the watchdog daemon won't be enabled.

5 Attack Implementation

To demonstrate the feasibility of remotely compromising a Polycom HDX system we were looking for a vulnerability which could be exploited in the most secure setup possible. Although the administrative web interface looks promising from a bug hunting perspective it should be firewalled in a production environment. The same is true for all the other administrative interfaces. However if the system is used for videoconferencing purposes either the H.323 or SIP port must be reachable.

In the following we concentrate on the H.323 protocol because this is what was configured in most of the videoconferencing equipment we encountered in the past.

5.1 Introduction to H.323

The H.323 standard is an umbrella recommendation from the ITU Telecommunications Standardization Sector (ITU-T). It defines multimedia collaboration on packet-based networks among two or more entities. Although originally intended for VoIP applications, nowadays it is mostly used for videoconferencing, while SIP displaced H.323 for most VoIP applications.

H.323 is a complex recommendation with many standards defining various aspects of the protocol. The most important signaling protocols from a bug hunting perspective are probably H.225.0-Q.931, H.225.0-RAS and H.245. H.225.0-Q.931 defines the call signaling protocol and media-stream packetization for setting up and releasing calls. H.225.0-RAS defines the procedures and signaling between endpoints and gatekeepers, while H.245 defines the procedures and signaling between two endpoints to exchange capabilities and control media streams.

All of these protocols are used for point-to-point calls and are used early in the connection setup phase, which makes them ideal candidates for potential vulnerabilities. For the purpose of this case study we are specifically interested in the H.225.0-Q.931 protocol. This protocol consists of different binary-encoded messages. Each binary encoded message consists of fields known as Information Elements which are encoded in ASN.1. Several different information elements are defined and provide various information to the remote site such e.g. the callers identity or other connection-related parameters. A complete list can be found in Table 4-3 of the Q.931 standard [4].

To initiate a call, a client establishes a connection on TCP port 1720 and sends a SETUP packet which indicates its desire to start a call. Several other packet types are part of a full call establishment, however we are only interested in the initial SETUP packet for the purpose of this discussion. Refer to [5] for a thorough introduction to H.323.

5.2 The Vulnerability

For every received SETUP packet the Polycom HDX system writes a call detail record (CDR) into its internal database. This even happens when the connection is not accepted. The CDR table is stored in a SQLite database which can be found in the `/data/polycom/cdr/new/localcdr.db` file on the HDX system.

For every call, several things are stored in the database. This includes the call start time, call end time, the call direction as well as the remote system name among other things. A complete list of stored items can be found in [10].

The remote system name stored in the CDR is directly taken from the string placed in the Display information element from the sent SETUP packet. However no input validation is performed on the string extracted from the packet which leads to two different vulnerabilities.

The SQL query string to insert a new CDR is constructed by simple string concatenation. Since the Display information element can contain strings with embedded single quote characters the code is vulnerable to a simple SQL injection vulnerability. Listing 4 shows the debug output after sending a single H.323 SETUP packet with a Display information element which contains a single quote character.

Looking promising at the first glance there is one complication. The constructed SQL statement is first passed to the `sqlite3_prepare_v2` API function which only compiles the first statement of the SQL query string and then passes the prepared statement to the `sqlite3_step` function. This prevents us from simply appending another SQL statement by using a semicolon. Although this might be exploitable we couldn't figure out a way in the limited time we spent looking into this vulnerability.

However there is another vulnerability related to the constructed SQL query string. After the string is constructed it is passed to the internal `puts()` function at `.text:1000c7c0` which ends up calling the `vsnprintf()` function inside `va_logmsg()` for logging purposes. The complete SQL query string is passed as the format string argument to `vsnprintf()` which leads to a format string vulnerability. Listing 5 shows the arguments passed to the `va_logmsg` function. Part of the `fmt` format string argument is the embedded Display information element which we control.

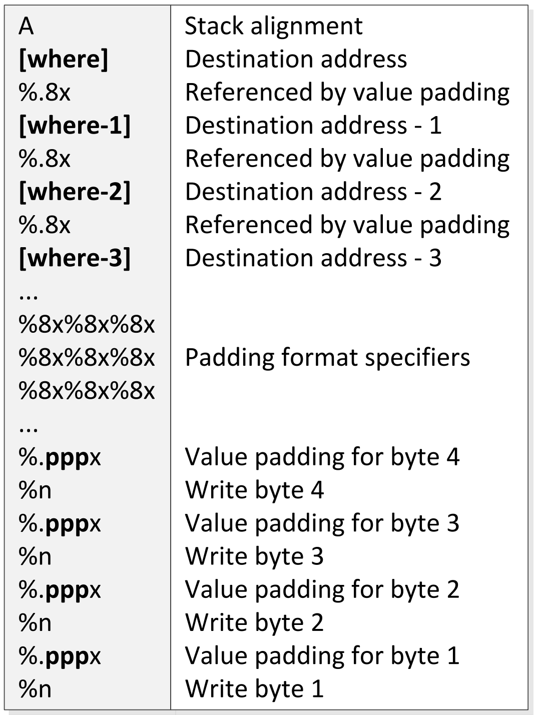
5.3 Exploitation

To exploit this bug basic format string exploitation techniques [13] can be used. However some constraints need to be considered which are described below. We start by defining our exploitation plan. We can turn the format string bug into a write4 primitive which then allows us to write four arbitrary bytes at an arbitrary address in memory. We then use the write4 primitive to first place our shellcode somewhere in memory and finally use it again to overwrite a function pointer triggering a jump into our shellcode.

5.3.1 Creating a Write4 Primitive

We first create a write4 primitive out of the bug. To write an arbitrary 32-bit value at an arbitrary memory address we are using four separate `%n` format specifiers to write four 32-bit values, decrementing the destination address for each written value. This way the written value will be composed of the least significant bytes of each written 32-bit value.

We add an appropriate number of padding characters between each `%n` format specifier to adjust the least significant byte of every written 32-bit value so that we control the final written value. The four pointers to the destination memory addresses are stored in the beginning of the string in decreasing order. They are followed by just enough dummy format specifiers so that the final `%n` format specifiers use the stored addresses as their destination pointers. The layout of the described Display information element string is shown in Figure 2. The "`.ppp`" strings represent the needed precision specifiers to adjust the value of each written byte.



A	Stack alignment
[where]	Destination address
%.8x	Referenced by value padding
[where-1]	Destination address - 1
%.8x	Referenced by value padding
[where-2]	Destination address - 2
%.8x	Referenced by value padding
[where-3]	Destination address - 3
…	
%8x%8x%8x	
%8x%8x%8x	Padding format specifiers
%8x%8x%8x	
…	
%.**ppp**x	Value padding for byte 4
%n	Write byte 4
%.**ppp**x	Value padding for byte 3
%n	Write byte 3
%.**ppp**x	Value padding for byte 2
%n	Write byte 2
%.**ppp**x	Value padding for byte 1
%n	Write byte 1

Figure 2: Format string layout for write4 primitive

A side effect of this technique is that we overwrite the three bytes in front of the written value. We just have to keep that in mind when using the write4 primitive.

With the created write4 primitive we can write four arbitrary bytes by sending a single H.323 SETUP packet. However there is a constraint on the characters we can use in the Display information element. Every byte must be larger 0 and smaller 0x80. Otherwise the format string bug is not triggered. Additionally the byte 0x25 representing the percent character is undesirable because it would be interpreted as part of a format specifier. This means for our write4 primitive that the addresses we can write to must only consist of those valid bytes. Additionally the three addresses `dest-3`, `dest-2` and `dest-1` preceding the destination address `dest` must also adhere to the constraint due to the way we implemented the write4 primitive.

```
DEBUG avc: pc[0]: INSERT into CDR_Table values('82','1347442631','1347443321',
'690','---','SQL'INJECT','','---','h323','0','','1','327','1','0','---','---',
'term
DEBUG avc: pc[0]: Can't prepare database: near "INJECT": syntax error
DEBUG avc: pc[0]: sqlInsert: time = 1
DEBUG avc: pc[0]: NOTIFY: SYS config cdrrowid1 0 "83" rw
DEBUG avc: pc[0]: H323Conn[0]: state:"incoming" --> "disconnecting"
DEBUG avc: pc[0]: H323Call[0]: hangup, cause code 16
```

Listing 4: Triggering SQL injection with a single H.323 SETUP packet

```
(gdb) break *0x1032E3AC
Breakpoint 1 at 0x1032e3ac: file ../../../src/Common/OS/logmsg.c, line 747.
(gdb) c
Breakpoint 5, 0x1032e3ac in va_logmsg (ap=0x5e97d298, level=<optimized out>,
    component=<optimized out>, fmt=0x5e97d344 "INSERT into CDR_Table values(
    '23','0','1347451282','1347451282','---','WE CONTROL THIS %n%n%n','','---',
    'h323','0','','1','365','1','0','---','---','terminal','','---','---',
    '---','---','---','---','The call has ended.','16','0','---','---','---',
    '---','---','---','---','---','---','---','---','---','25');")
    at ../../../src/Common/OS/logmsg.c:747
```

Listing 5: Format string vulnerability in SQL query string logging code

5.3.2 Storing the Shellcode

The next step is to find a place in memory where we can store our shellcode. Having the address constraint in mind a quick look through the program segments turns up only a single segment which is writeable and fulfills our constraint. It's the BSS segment which ranges from the address `0x10d48320` to `0x11ea39a0`. Various memory blocks of consecutive 80 bytes can be found in the BSS segment which is enough to store a simple shellcode.

5.3.3 Function Pointer Overwrite

Next we need to find a function pointer which can be overwritten and triggered to jump into our shellcode. The `mtctr` and `bctrl` PowerPC instructions are used to perform indirect function calls. In order to find potential candidates we look at the cross-references of all variables in the BSS segment which are located at addresses fulfilling the mentioned address constraint and check if they are used to store a function pointer.

One of the first variables found is the `CodecPoleList` variable stored at the address `0x111d0918`. This variable stores a pointer which is referenced in the `VideoBitStreamPoleTimerProc` function. The `CodecPoleList` pointer plus a constant offset of 0x1494 is dereferenced and then used to read a function pointer where the code jumps to. We verified that this function is called every few seconds making it a perfect target for the overwrite.

5.3.4 Shellcode

For the proof-of-concept exploit we use a basic shellcode which just calls the `system()` library function with a shell command line of our choosing. The shellcode, a pointer to the shellcode and the shell command line are first stored in the BSS segment by triggering the format string bug several times and finally we overwrite the `CodecPoleList` pointer with the address where we stored the pointer to our shellcode minus the constant 0x1494. This immediately triggers the execution of the shellcode and executes our shell command line as the user `root`, which is the user the `avc` process is running under. Since the `VideoBitStreamPoleTimerProc` function is running in a separate timer thread the system keeps running while the payload is executed via the `system()` API function.

5.3.5 Reliability

The created proof-of-concept exploit was only used to demonstrate the feasibility of remote code execution and no further work has been done to make it one-hundred percent reliable. There are two influencing factors we know of which could make the exploit fail and would require another attempt.

The SQL query string for the CDR entry starts with an increasing ID. We only have control over the string after this ID. The ID increases for every record written to the database, but wraps around to 0 when it reaches the value 99. This means it can either be a one-digit or two-digit number. Since our padding relies on the number of characters preceding the part of the string we control, we are making an educated guess and assume a two-digit number. If the guess was wrong or the ID wrapped around during the exploit attempt, the system will crash and reboot. After the system rebooted we can send ten dummy SETUP packets to make the ID a two-digit number. If the crash happened due to this problem, a second exploit attempt will succeed.

A second unreliability stems from another increasing ID embedded in the SQL query string after the part which we control. It's the 13th column value which can grow up to a value of around 2411 after which it will wrap around. The length of this ID influences the bytes we are writing with our write4 primitive. The range of possible lengths is one to four with increasing probability. Again we make an educated guess about the length of the ID and just try another one if the system crashes and reboots.

Again the mentioned problems can probably be worked around with a bit more effort. However we didn't invest the time to improve the reliability, since the exploit in its current form proofs that remote code execution is possible. Additionally the system automatically reboots once a crash is encountered giving the attacker multiple tries.

6 Payload Development

After compromising an HDX system you typically want to control the device's peripherals like moving the camera, recording voice using the microphone, playing sounds or displaying childish messages on the screen. To find out how to perform those actions the Polycom Command Shell implemented in the `/opt/polycom/bin/psh` binary is a good starting point because it is possible to perform most of these actions via the provided command line interface.

Reverse engineering the `psh` binary shows that the functionality itself is not implemented inside the `psh` binary. The Polycom Command Shell communicates via the Polycom XCOM IPC mechanism with the Java component which implements most of the functionality. So by talking XCOM directly our payload can make use of all the functionality.

6.1 XCOM Protocol

The XCOM protocol is a simple text-based protocol locally provided on port 4121. In addition to the TCP communication channel it also provides an asynchronous data reception mechanism via UNIX domain sockets.

Commands and responses send and received on the TCP socket are prefixed with an uppercase character identifying the command or response class, followed by a colon and space character. Every sent command is answered with a single line response. Notifications (prefixed by "N: ") can be received asynchronously by an XCOM client without first sending a command.

When the `psh` binary is started it first creates a UNIX domain socket in the `/tmp` directory which acts as its asynchronous listener. It then connects to the XCOM server on port 4121 and sends a command to register the created async listener with the current XCOM session. The first part of the command is the desired XCOM session type which is identical to the value which can be provided to the `psh` binary via the `-t` command line switch. It can be either "`api`" for an API session or "`telnet`" for a Polycom Command Shell session. The second part is the path of the previously created UNIX domain socket where the XCOM server sends asynchronous responses to executed commands. The last part of the command is the current pseudo terminal.

After registering its asynchronous listener `psh` authenticates the current user. It first reads the username and password and sends them in hex-encoded form to the XCOM server for verification. If authentication was successful the user's role is returned (e.g. "adminlogin" for a successful administrator login) on the bound asynchronous listener. If authentication failed, the string "failed" is returned. The XCOM communication of a failed administrator login can be seen in Listing 6. The used `async` binary is just a small tool we wrote which creates a UNIX domain socket and prints out any data it receives.

Authentication is only a service used by the `psh` binary, but it's not required in order to use the XCOM interface. In order to send arbitrary commands to the XCOM server we just have to register an asynchronous listener (which we don't have to create) and then we can send all the available Polycom Command Shell commands by prefixing them with the "C: " string prefix. A listing of all `psh` commands can be obtained with the `help` command in the Polycom Command Shell. The `help` command can also be used to get detailed usage instructions for each command. Listing 7 shows how to move the camera by locally connecting to the XCOM server via telnet.

In addition to the commands already provided by the `psh` binary, further commands can be found by reverse engineering the relevant Java classes implementing the XCOM server. With the help of the commands provided by the XCOM interface it would be possible to create a HDX-specific rootkit which could take complete control over the attached video camera and microphones.

6.2 Framebuffer Drawing

If the goal of the attack is to control the content of the screen (we only used this for demo purposes), the

```
# ./async /tmp/listener &
[*] Connecting to async listener /tmp/listener...
[*] Successfully established connection
# telnet localhost 4121
R: telnet /tmp/listener /dev/pts/0
R: 0
E: _control login "61646d696e" "666f6f626172" /dev/pts/0
[*] Received on async listener: failed
R: 0
```

Listing 6: XCOM communication of a failed administrator login

```
# telnet localhost 4121
R: telnet /tmp/dummy /dev/pts/0
R: 0
C: camera near move up
N: SYS+config+powerlight+0+%22Blue*on*0*0%22+rw
N: VID+videoroute+set+27+complete+vout1+1920+1080+Component+50+Interlaced
N: VID+videoroute+set+28+complete+mon3+704+576+SVideo+25+Interlaced
R: 0
```

Listing 7: Moving the camera via XCOM with telnet

system provides a separate binary which can be used for this purpose. The `frame-buf-draw` binary allows drawing on the framebuffer. It's possible to draw arbitrary JPEG images or text. The binary itself can be found in the `/opt/polycom/bin/post.tgz` archive which must be unpacked first. Usage instructions are shown in Listing 8.

7 Related Work

A sheer number of publications have been released in the past on the security of the typical signaling (H.323 and SIP) and media transfer (RTP) protocols used for videoconferencing. Implementation flaws in the signaling protocols are covered in [7], [6], [15] and [14], to just name a few. Most of the work in that area was published in the context of VoIP security though.

The publication [1] talks about securing Polycom videoconferencing endpoints. However only aspects concerning the secure configuration are covered. Quite recently HD Moore demonstrated the need for securing the configuration of videoconferencing systems in [2]. He showed that thousands of videoconferencing systems were publicly accessible over the internet and had the call auto-answer feature turned on. He found systems at several top venture capital and law firms, pharmaceutical and oil companies and courtrooms. By calling into those systems he was able to listen into conversations and remotely control the camera.

8 Conclusion

In this paper we have laid the groundwork for conducting future security assessments on the HDX series of devices which is one of the main product lines of videoconferencing equipment sold by the market leader Polycom. After analyzing the software update file format and showing how to get system level access to the otherwise closed devices, we described how to setup a vulnerability development environment which can be used for bug hunting and exploit development.

Based on that, we demonstrated the feasibility of remote exploitation by implementing an exploit for one of the vulnerabilities we identified in the H.323 stack affecting the latest software version. This allowed us to compromise Polycom HDX systems remotely by just sending H.323 SETUP packets.

Finally we discussed possible exploit payloads and introduced Polycom's XCOM IPC mechanism which could be used to control connected peripherals such as the video camera and microphones in order to implement surveillance functionality in a post-exploitation scenario.

References

[1] Christianson, J. S. Polycom Videoconferencing Endpoint Security and Configuration. SANS Institute, InfoSec Reading Room, `http://www.sans.org/reading_room/whitepapers/commerical/polycom-videoconferencing-endpoint-security-configuration_21` (2012/10/29), 2003.

```
# /usr/diags/video/bin/frame-buf-draw --help
usage: frame-buf-draw [ option ... ] action

Options are:
  --help              Show this text.
  --version           Show program version.
  --left <num>        Left pixel offset of drawing.  Default is 0.
  --top <num>         Top pixel offset of drawing.  Default is 0.
  --right <num>       Right pixel offset of black rectangle.
                      Default is edge of monitor.
  --bottom <num>      Bottom pixel offset of black rectangle.
                      Default is edge of monitor.
  --init              Initialize frame buffer driver before using.
  --file <file_name>  Image file to read.

Actions are:
  --black             Draw a solid black rectangle.
  --image             Draw an image from a file.
  --progdot <num>     Assume image file is a progress dot, and draw
                      it at the location for dot index <num> instead
                      of --left,--top.
  --text <num>        Assume image file is a text message, and draw
                      it at the location for row index <num> instead
                      of --left,--top.
  --progerase <num>   Same as --black, but assume upper left corner
                      is row index <num> instead of --left,--top,
                      where row zero is the dots.
  --maybe-erase       Erase the entire screen if the splash image
                      consumes too much of it.
  --ipaddr <string>   Draw decimal IP address <string> at the
                      location for row index zero.
```
Listing 8: Usage information for the `frame-buf-draw` program

[2] HD Moore. Board Room Spying for Fun and Profit. https://community.rapid7.com/community/metasploit/blog/2012/01/23/video-conferencing-and-self-selecting-targets (2012/10/29), January 2012.

[3] Heffner, C. Binwalk. http://code.google.com/p/binwalk/ (2012/10/30).

[4] International Telecommunication Union. Isdn user-network interface layer 3 specification for basic call control. ITU-T Recommendation Q.931, 1988.

[5] Kumar, V., Korpi, M., and Sengodan, S. *IP Telephony with H.323: Architectures for Unified Networks and Integrated Services*. John Wiley & Sons, Inc., 2001.

[6] N.N.P. VoIPER: Finding VoIP vulnerabilities while you sleep. Presentation at DEF CON 16, August 2008.

[7] Nuwere, E., and Varpiola, M. The Art of SIP fuzzing and Vulnerabilities Found in VoIP. Presentation at Black Hat Japan, October 2005.

[8] Office of the Press Secretary. Readout of the President's Videoconference with Chancellor Merkel, President Hollande and Prime Minister Monti. http://www.whitehouse.gov/the-press-office/2012/05/30/readout-presidents-videoconference-chancellor-merkel-president-hollande- (2012/10/29), May 2012.

[9] Polycom. Solutions by Industry. http://www.polycom.com/solutions/solutions-by-industry.html (2012/11/05).

[10] Polycom. Administrator's Guide for Polycom HDX Systems. http://supportdocs.polycom.com/PolycomService/support/global/documents/support/setup_maintenance/products/video/hdx_ag.pdf (2012/10/30), July 2012.

[11] Polycom. Integrator's Reference Manual for Polycom HDX Systems. http://supportdocs.polycom.com/PolycomService/support/global/documents/support/setup_maintenance/products/video/hdx_irm.pdf (2012/10/30), July 2012.

[12] Social Security Administration. Why You Should Have Your Hearing By Video. SSA Publication No. 70-067, ICN 443300, http://www.socialsecurity.gov/appeals/odar_pubs/70-067.html (2012/10/29), April 2008.

[13] team teso, S. . Exploiting format string vulnerabilities. http://packetstormsecurity.org/papers/unix/formatstring-1.2.tar.gz [2012/11/02].

[14] Trammell, D. D. VoIP Attacks! Presentation at CSI 2007, http://druid.caughq.org/presentations/VoIP-Attacks.pdf (2012/10/29), 2007.

[15] University of Oulu. PROTOS Test-Suite: c07-h2250v4. https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c07-h2250v4 (2012/10/29).