Security Flaws in Universal Plug and Play

Unplug. Don't Play.

January 2013

HD Moore

2013-01-29 v1.0.0

Executive Summary

Universal Plug and Play (UPnP) is a protocol standard that allows easy communication between computers and network-enabled devices. This protocol is enabled by default on millions of systems, including routers, printers, media servers, IP cameras, smart TVs, home automation systems, and network storage servers. UPnP support is enabled by default on Microsoft Windows, Mac OS X, and many distributions of Linux.

The UPnP protocol suffers from a number of basic security problems, many of which have been highlighted over the last twelve years. Authentication is rarely implemented by device manufacturers, privileged capabilities are often exposed to untrusted networks, and common programming flaws plague common UPnP software implementations. These issues are endemic across UPnP-enabled applications and network devices.

The statistics in this paper were derived from five and a half months of active scanning. UPnP discovery requests were sent to every routable IPv4 address approximately once a week from June 1 to November 17, 2012. This process identified over 81 million unique IP addresses that responded to a standard UPnP discovery request. Further probes determined that approximately 17 million of these systems also exposed the UPnP Simple Object Access Protocol (SOAP) service to the world. This level of exposure far exceeded the expectations of the researchers.

This paper quantifies the exposure of UPnP-enabled systems to the internet at large, classifies these systems by vendor, identifies specific products, and describes a number of new vulnerabilities that were identified in common UPnP implementations. Over 1,500 vendors and 6,900 products were identified that are vulnerable to least one of the security flaws outlined in this paper. Over 23 million systems were vulnerable to a single remote code execution flaw that was discovered during the course of this research.

Rapid7 worked with CERT/CC to notify the open source projects and device manufacturers vulnerable to the issues described in this paper. Unfortunately, the realities of the consumer electronics industry will leave most systems vulnerable for the indefinite future. For this reason, Rapid7 strongly recommends disabling UPnP on all internet-facing systems and replacing systems that do not provide the ability to disable this protocol.

Rapid7 has provided a number of tools to help identify UPnP-enabled systems, including the free ScanNow for UPnP, modules for the open source Metasploit Framework, and updates to the Nexpose vulnerability management platform.

Summary Statistics

2.2% of public IPv4 addresses respond to UPnP discovery requests from the internet.

81 million unique IP addresses respond to UPnP discovery requests, slightly more than all IPs allocated to Canada.

20% of those 81 million systems also expose the SOAP API to the internet at large. This service can allow an attacker to target systems behind the firewall.

4 software development kits account for 73% of all discovered UPnP instances.

332 products use MiniUPnPd version 1.0, which is remotely exploitable. Over 69% of all MiniUPnPd fingerprints were version 1.0 or older.

23 million fingerprints match a version of libupnp that exposes the system to remote code execution.

1 UDP packet is all it takes to exploit any of the 8 newly-discovered libupnp vulnerabilities. This packet can be spoofed.

Immediate Actions

Given the high level of exposure and the potential impact of a successful attack, Rapid7 strongly recommends that UPnP be disabled on all external-facing systems and devices providing a critical function.

Internet Service Providers

ISPs should review any equipment that they are providing to subscribers to verify that UPnP is not exposed on the WAN interface.

If the equipment is affected, one of the following solutions should be considered:

- Pushing a configuration update that disables UPnP across the subscriber base
- Pushing a software update that removes UPnP capabilities from the device
- Replacing customer equipment with a device that can be configured securely
- Implementing network-wide ACLs for UDP port 1900 and specific TCP ports

Businesses

Companies should verify that all external-facing devices do not expose UPnP to the internet. Rapid7 provides ScanNow UPnP as well as Metasploit modules that can detect vulnerable UPnP services. If any equipment is found that exposes UPnP, the best option is to disable the service, and if that is not possible, replace the device with a model that allows this.

Many network devices inside of the firewall are UPnP-enabled. Examples include network printers, IP cameras, storage systems, and media servers. Any devices found that expose UPnP should be reviewed for potential security impact. If equipment is found to use a vulnerable UPnP implementation, the vendor should be contacted to determine their timeframe for an update. If the UPnP service cannot be disabled and the vendor does not have an update, it may be prudent to segment the device from the rest of the network.

Home and Mobile Users

Home and mobile PC users should ensure that the UPnP function on their home routers and mobile broadband devices has been disabled. The free ScanNow UPnP tool from Rapid7 can help identify affected devices. If the device does not provide the ability to disable UPnP, the company that makes or sells the device should be contacted to see if an update is available that provides this capability. Worst case, users should replace vulnerable equipment with devices that do not support UPnP, or at least ones that provide the ability to disable it.

Contents

Introduction

Universal Plug and Play (UPnP) is a protocol standard that allows easy communication between computers and network-enabled devices. This protocol is enabled by default on millions of systems, including routers, printers, media servers, smart TVs, and network storage servers. UPnP support is enabled by default on Microsoft Windows, Mac OS X, and many distributions of Linux.

The UPnP protocol suffers from a number of basic security problems, many of which have been highlighted over the last twelve years. Authentication is rarely implemented by device manufacturers, privileged capabilities are often exposed to untrusted networks, and common programming flaws plague common UPnP software implementations. These issues are endemic across UPnP-enabled applications and network devices.

This paper focuses on three specific classes of problems

- Programming flaws in common UPnP discovery protocol (SSDP) implementations can be exploited to crash the service and execute arbitrary code.
- Exposure of the UPnP control interface (SOAP) exposes private networks to attacks from the internet at large and can leak sensitive data.
- Programming flaws in the UPnP HTTP and SOAP implementations can be exploited to crash the service and execute arbitrary code.

Acknowledgements

Jared Allar of CERT/CC did an amazing job managing the disclosure process, notifying affected vendors, and directly contributing to the libupnp patch. JPCERT/CC was invaluable in their facilitation of vendor coordination with KrCERT/CC and CNCERT/CC. Thomas Bernard of the MiniUPnP project was especially helpful in providing feedback on this draft. Kurt Seifried of the Red Hat Security Response team was extremely helpful, especially at 3:00am.

Universal Plug and Play

The Universal Plug and Play (UPnP) protocol suite was designed to simplify the discovery and control of network devices. Capabilities vary by device, but include control of incoming port mappings on home routers, easy identification of network printers, and the management of media services such as streaming video. Applications use UPnP to access and configure network-connected services. For example, a BitTorrent application may use UPnP to identify the network's public IP address and automatically forward incoming connections on the router to the computer running the BitTorrent software. Another example is the "Add Device"

wizard in Microsoft Windows. This wizard uses the UPnP protocol to identify network devices such as scanners and printers on the local network. UPnP supports the discovery of these devices as well as the ability to manage them, enabling tasks such as clearing the print queue or initiating a scan.

The UPnP suite consists of two distinct services. The Simple Service Discovery Protocol (SSDP) service listens on UDP port 1900 and is responsible for advertising available services and responding to discovery requests.

The second component of UPnP is the HTTP service. During the discovery process, the SSDP protocol is used to identify the location of the UPnP HTTP service and service description file for a given system. The service description file is an XML document hosted by the UPnP HTTP service. The TCP port used by this service varies by vendor and is often chosen at random and the SSDP discovery response indicates the current location of this service.

In addition to providing the service description file, the HTTP service also hosts the Simple Object Access Protocol (SOAP) interface. SOAP provides a way for other systems on the network to call a defined set of functions. All useful UPnP functionality is implemented through SOAP API calls which are further categorized into device profiles. Each profile implements a number of services, which, in turn, expose a set of actions and state variables. Commonly supported device profiles include the Internet Gateway Device suite, Printer, and MediaServer.

A system that wishes to use a UPnP-enabled device would first send a SSDP M-SEARCH request to the local network. Systems that run a local SSDP service may simply query the existing device cache. The response would indicate the HTTP location of the device description XML. This XML provides detailed device information and a list of supported services. Once the device description has been obtained and a list of services found, another request is sent for the service description XML, which defines the actions available, what parameters they take, and the format of the response. Finally, the application would send a SOAP request for the specific URL of the target service, specifying the SOAP action in the header of the HTTP request and supplying input parameters as an XML document in the body. The device would execute the request and return the response as another XML document.

Research Results

This paper is the result of a research project spanning the second half of 2012. The goal of this research was to quantify the number of UPnP-enabled systems that expose the SSDP service to the internet, determine how many of those also exposed the SOAP service, and identify security weaknesses in the most commonly used UPnP implementations.

Widespread Exposure of the UPnP SSDP Service

The UPnP SSDP service is designed to allow device discovery over the local network. This service exposes version information and network configuration details over UDP port 1900. Our findings indicate that the SSDP service has been misconfigured across thousands of products, resulting in widespread exposure of the SSDP service to the internet at large. Over 81 million unique IPs were found to expose the SSDP service to the public internet.

Widespread Exposure of the UPnP SOAP Service

The UPnP SOAP service provides access to device functions that should not be allowed from untrusted networks, including the ability to open holes through the firewall. Our findings indicate that the SOAP service has been misconfigured by over 1,500 network device vendors and 6,900 devices, resulting in widespread exposure of the SOAP service to the internet at large. We estimate that approximately 17 million unique IPs expose the SOAP service to the public internet.

UPnP Exposure Concentrated Across Four Implementations

Over 73% of all UPnP instances discovered through SSDP were derived from only four software development kits. These include the Portable SDK for UPnP Devices, MiniUPnP, a commercial stack that is likely developed by Broadcom, and another commercial kit that could not be tracked to a specific developer. This heavy concentration substantially increases the impact of any vulnerabilities found within these implementations.

Network Devices use Outdated UPnP Implementations

The UPnP instances found to be using the Portable SDK for UPnP Devices and the MiniUPnP library both expose the software library version in the SSDP response. Our

analysis of these versions indicates that the majority of exposed devices are using UPnP libraries that are over four years old. In many cases, even devices that were recently manufactured were still using outdated UPnP software libraries.

Exploitable Vulnerabilities in the Portable SDK for UPnP Devices SSDP Parser

The Portable SDK for UPnP devices was found to have no less than eight remotely exploitable vulnerabilities in the SSDP parser. Although most of the systems found to be running this software were using an obsolete version of this library, even the latest version at the time of this research was vulnerable to two remote stack overflows.

Over 25% of all exposed SSDP services are using this software (approximately 23 million systems). These issues have been assigned the following CVEs:

CVE-2012-5958	**CVE-2012-5959**	**CVE-2012-5960**	**CVE-2012-5961**
CVE-2012-5962	**CVE-2012-5963**	**CVE-2012-5964**	**CVE-2012-5965**

The vulnerabilities assigned to CVE-2012-5958 and CVE-2012-5959 affect all versions of the Intel SDK and all Portable SDK versions prior to 1.6.18.

Exploitable Stack Overflow in the MiniUPnP SOAP Handler

The MiniUPnP library was found to have a remotely exploitable stack overflow in the SOAP handler. This vulnerability was fixed in version 1.1, but over 14% of all exposed SSDP services advertised MiniUPnP version 1.0. This vulnerability is somewhat limited in that an attacker must be able to access the SOAP endpoint. Over 330 products were identified that used an old version of MiniUPnP and exposed the SOAP endpoint to the public internet. This issue has been assigned **CVE-2013-0230**.

Denial of Service Flaws in the MiniUPnP SSDP Parser

The MiniUPnP library was found to have a number of parsing flaws in the SSDP handler. These vulnerabilities were fixed in version 1.4 and could allow a remote attacker to terminate the process that embeds the MiniUPnP library. The issues have been assigned **CVE-2013-0229**.

Methodology

The statistics in this paper were derived from data collected over a period of five and a half months (June 1 to November 17, 2012). A UPnP SSDP M-SEARCH request was sent to every IPv4 address on the internet approximately once a week. Over 93 million unique fingerprints were obtained as the combination of IP address and UPnP SSDP server version. These fingerprints covered a total of 81 million unique IP addresses. Although some fingerprints are more useful than others, many of these returned information about the OS, the name of the UPnP library, and often the version of this library. A typical string from the Server header of the SSDP response is shown below.

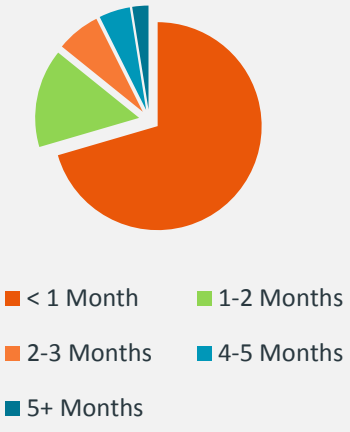
```
Linux/2.4.31_mvl31, UPnP/1.0, Intel SDK for UPnP devices/1.3.1
```

In addition to the Server header, the Location header returned in the SSDP response provides three key pieces of information. The IP address shown in this URL should correspond to the internal IP of a router, but often does not. The port number and device descriptor path can also serve as additional methods to fingerprint specific devices.

A follow-up scan was launched in December of 2012 in order to determine how many UPnP-enabled systems exposed the SOAP interface to the internet. This scan also identified specific products and vendors through the advertised device description XML page. To conduct this scan, 150,000 class C (/24) subnets were selected at random from 2.2 million class C subnets containing UPnP responsive systems. These 150,000 subnets represented a search space of 38.4 million IP addresses, all of which were probed for exposed UPnP SOAP endpoints. This resulted in just over 1 million device fingerprints. Extrapolating this ratio across all 2.2 million subnets yields an estimate of 17.25 million IPs that expose the UPnP SOAP endpoint to the internet. Spot checks of additional subnets confirm this ratio.

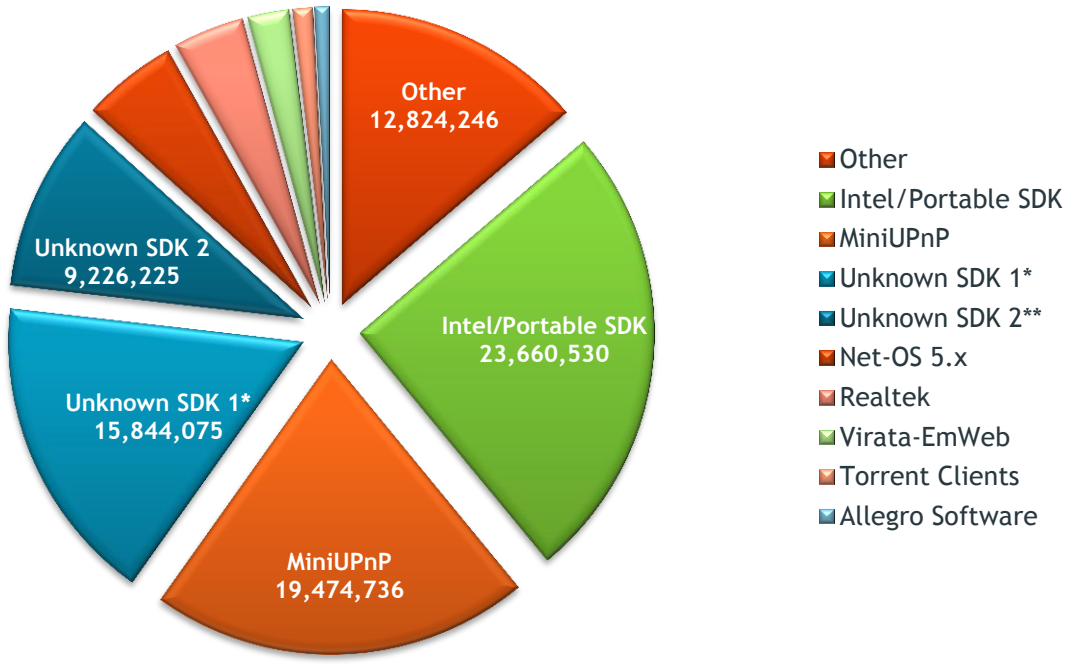
In order to determine how frequently these systems changed IP address, the entire SSDP fingerprint database was analyzed at monthly intervals and a mismatch of an IP between months was used to detect that the IP had changed.

Over 65 million of those 81 million unique IP addresses responded with the same UPnP server fingerprint for less than one month. Given the high rate of churn, the relative values of the statistics in this section are more important than the absolute values.



Data Distribution

The chart below shows the distribution of 93 million unique fingerprints across the known UPnP libraries and development kits. The key finding is that a fairly small number of libraries represent the majority of exposed UPnP implementations.

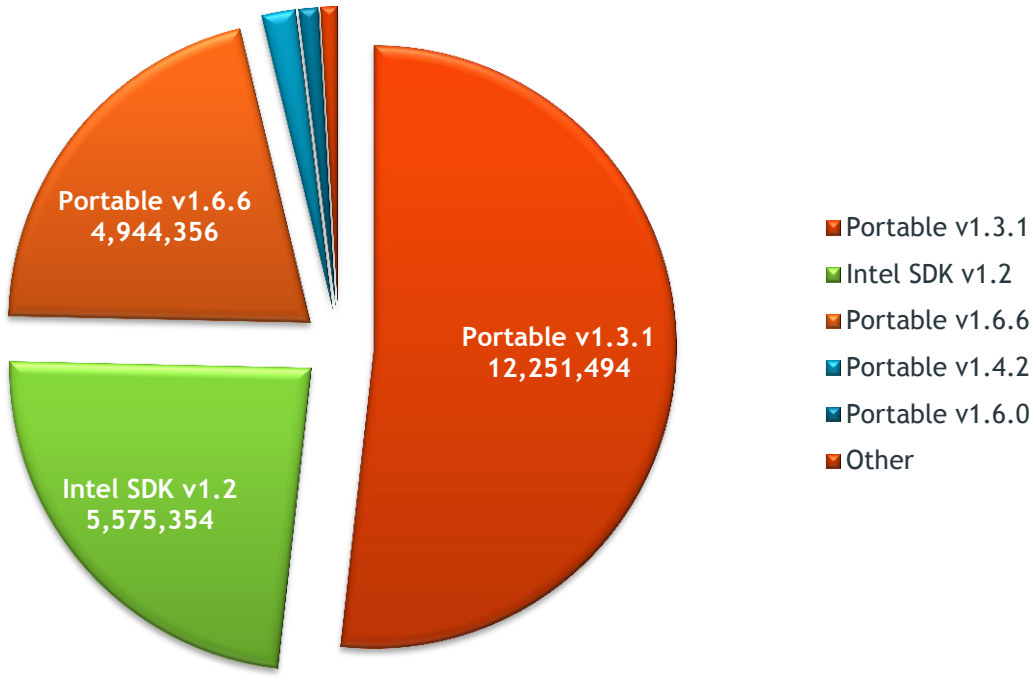


The vendor marked *Unknown SDK 1* is proprietary stack used in devices branded by Broadcom, D-Link, TP-Link, and many other vendors. This stack does not explicitly identify itself. This SDK uses the generic string of "Custom/1.0 UPnP/1.0 Proc/Ver". A common trait of this SDK is to use the fixed TCP port 5431 for the HTTP listener.

The vendor marked *Unknown SDK 2* is used by a mix of smaller networking device vendors and has proven difficult to identify. This SDK uses the generic string of "System/1.0 UPnP/1.0 IGD/1.0". A common trait of this SDK is to use TCP port 80 for the HTTP listener. The HTTP listener is rarely reachable from the internet at large, which makes more detailed fingerprinting difficult.

Version Distribution: Portable SDK for UPnP Devices

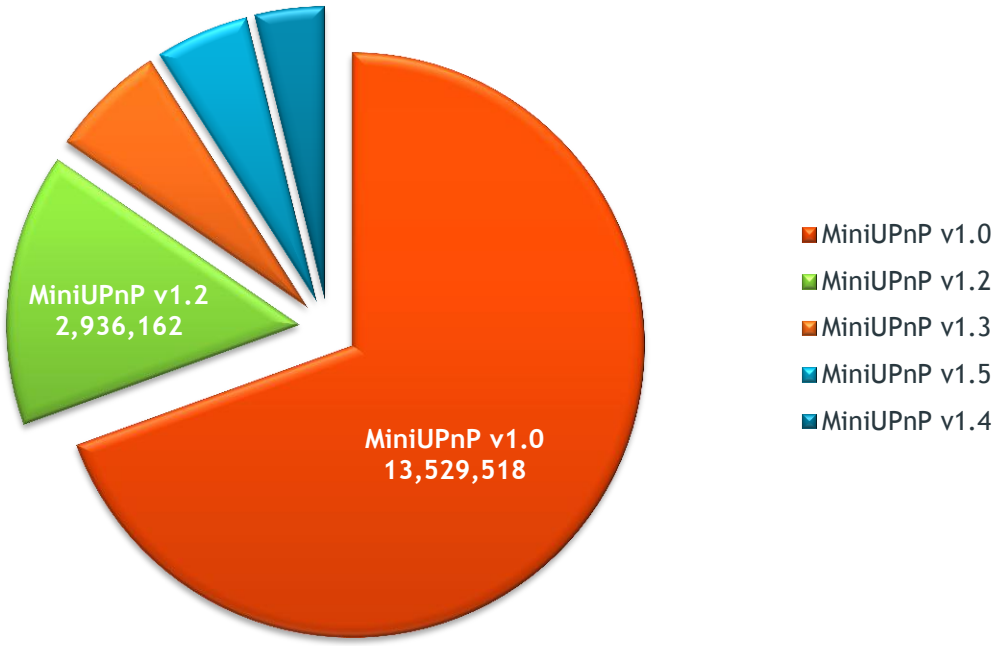
Over 25% of the identified fingerprints (23.6 million) matched UPnP implementations based on the Intel SDK for UPnP devices, a reference implementation that has since become the Portable SDK for UPnP Devices (libupnp). The version split of these 23.6 million fingerprints is shown below.



Over 12 million fingerprints are for version 1.3.1 of the Portable SDK for UPnP devices, while another 5.5 million use the original Intel SDK v1.2 reference implementation.

Version Distribution: MiniUPnP

In second place based on frequency, the MiniUPnP software library represents over 21% of all unique fingerprints. The version split of these 19.4 million fingerprints is shown below.



Over 13 million fingerprints are for version 1.0 MiniUPnP. This data is surprising given that the MiniUPnP software didn't reach version 1.0 until January of 2008. Although version 1.1 was released in April of 2008, it is not represented at all in the SSDP responses.

One possible explanation is that many of the systems fingerprinted as 1.0 were actually running a pre-release build that still identifies as that version. A number of beta versions and release candidates were released between 2005 and 2008, all of which report version 1.0 through the SSDP and SOAP services. This was confirmed with the MiniUPnP developer.

Vulnerabilities

Intel/Portable SDK for UPnP Devices (libupnp)

The majority of identified UPnP implementations are based on the Intel SDK for UPnP Devices. The Intel SDK was created as a reference implementation and has transitioned to an open source project under the Portable SDK for UPnP Devices name, or simply “libupnp”.

The key thing to note is how old this code base is. The original Intel SDK was released in 2001 and the most prevalent version of the Portable SDK (1.3.1) was released in 2006. The third-most common version (1.6.6) is still over four years old at the time of this writing.

Given the age of this code and the number of devices on which it is installed, one would assume it had been audited for security flaws at some point in the past. Prior to this research, not a single CVE identifier or OSVDB entry had been created for an identified flaw in the libupnp software. Keep in mind that almost a quarter of the libupnp-based systems were running a version of this SDK that is over a decade old and more than half are using a version of libupnp that is at least six years old.

Simple Service Discovery Protocol Vulnerabilities

Eight vulnerabilities were identified during the audit of the libupnp code base. Not all of these flaws apply to each version, but CVE-2012-5958 and CVE-2012-5959 trigger an exploitable condition in the SSDP parser on all versions of the Intel SDK and all versions of the Portable SDK up to version 1.6.18.

In version 1.3.1 of the Portable SDK for UPnP Devices, the SSDP parser code lives within *upnp/src/ssdp/ssdp_server.c*. The *unique_service_name()* function is responsible for parsing an incoming SSDP request and filling in the fields of structure representing the event. In the code listed below, obvious security flaws have been highlighted in yellow. Note that the **cmd** variable points to the string received by the remote end. The **Evt** structure contains fixed-length string buffers and is stored on the stack by a calling function.

```
int
unique_service_name( IN char *cmd,
                     IN SsdpEvent * Evt )
{
    char *TempPtr,
      TempBuf[COMMAND_LEN],
     *Ptr,
     *ptr1,
     *ptr2,
     *ptr3;
    int CommandFound = 0;

    if( ( TempPtr = strstr( cmd, "uuid:schemas" ) ) != NULL ) {
```

```
    ptr1 = strstr( cmd, ":device" );
    if( ptr1 != NULL ) {
        ptr2 = strstr( ptr1 + 1, ":" );
    } else {
        return -1;
    }

    if( ptr2 != NULL ) {
        ptr3 = strstr( ptr2 + 1, ":" );
    } else {
        return -1;
    }

    if( ptr3 != NULL ) {
        sprintf( Evt->UDN, "uuid:%s", ptr3 + 1 );
    } else {
        return -1;
    }

    ptr1 = strstr( cmd, ":" );
    if( ptr1 != NULL ) {
        strncpy( TempBuf, ptr1, ptr3 - ptr1 );
        TempBuf[ptr3 - ptr1] = '\0';
        sprintf( Evt->DeviceType, "urn%s", TempBuf );
    } else {
        return -1;
    }
    return 0;
}

if( ( TempPtr = strstr( cmd, "uuid" ) ) != NULL ) {
    //printf("cmd = %s\n",cmd);
    if( ( Ptr = strstr( cmd, "::" ) ) != NULL ) {
        strncpy( Evt->UDN, TempPtr, Ptr - TempPtr );
        Evt->UDN[Ptr - TempPtr] = '\0';
    } else {
        strcpy( Evt->UDN, TempPtr );
    }
    CommandFound = 1;
}

if( strstr( cmd, "urn:" ) != NULL
    && strstr( cmd, ":service:" ) != NULL ) {

    if( ( TempPtr = strstr( cmd, "urn" ) ) != NULL ) {
        strcpy( Evt->ServiceType, TempPtr );
        CommandFound = 1;
    }
}

if( strstr( cmd, "urn:" ) != NULL
    && strstr( cmd, ":device:" ) != NULL ) {
    if( ( TempPtr = strstr( cmd, "urn" ) ) != NULL ) {
        strcpy( Evt->DeviceType, TempPtr );
        CommandFound = 1;
    }
}

if( CommandFound == 0 ) {

    return -1;
}
```

CVE-2012-5961

CVE-2012-5958

CVE-2012-5962

CVE-2012-5959

CVE-2012-5963

CVE-2012-5964

CVE-2012-5965

```
    return 0;
}
```

There are no less than seven unique buffer overflows in version 1.3.1 of this code, all of which can be used to corrupt the stack and potentially execute arbitrary code. In most instances, the application linked to the libupnp library runs as the root user account, since the actions handled through the SOAP API often require it (managing firewall rules, etc).

Fast forwarding to the latest libupnp released at the time of this writing, version 1.6.17:

```
int unique_service_name(char *cmd, SsdpEvent *Evt)
{
	char TempBuf[COMMAND_LEN];
	char *TempPtr = NULL;
	char *Ptr = NULL;
	char *ptr1 = NULL;
	char *ptr2 = NULL;
	char *ptr3 = NULL;
	int CommandFound = 0;
	size_t n = (size_t)0;

	if (strstr(cmd, "uuid:schemas") != NULL) {
		ptr1 = strstr(cmd, ":device");
		if (ptr1 != NULL)
			ptr2 = strstr(ptr1 + 1, ":");
		else
			return -1;
		if (ptr2 != NULL)
			ptr3 = strstr(ptr2 + 1, ":");
		else
			return -1;
		if (ptr3 != NULL) {
			if (strlen("uuid:") + strlen(ptr3 + 1) >= sizeof(Evt->UDN))
				return -1;
			snprintf(Evt->UDN, sizeof(Evt->UDN), "uuid:%s",
				ptr3 + 1);
		}
		else
			return -1;
		ptr1 = strstr(cmd, ":");
		if (ptr1 != NULL) {
			n = (size_t)ptr3 - (size_t)ptr1;
			strncpy(TempBuf, ptr1, n);
			TempBuf[n] = '\0';
			if (strlen("urn") + strlen(TempBuf) >= sizeof(Evt->DeviceType))
				return -1;
			snprintf(Evt->DeviceType, sizeof(Evt->DeviceType),
				"urn%s", TempBuf);
		} else
			return -1;
		return 0;
	}
	if ((TempPtr = strstr(cmd, "uuid")) != NULL) {
		if ((Ptr = strstr(cmd, "::")) != NULL) {
			n = (size_t)Ptr - (size_t)TempPtr;
			strncpy(Evt->UDN, TempPtr, n);
			Evt->UDN[n] = '\0';
		} else {
```

CVE-2012-5958

CVE-2012-5959

```
                        memset(Evt->UDN, 0, sizeof(Evt->UDN));
                        strncpy(Evt->UDN, TempPtr, sizeof(Evt->UDN) - 1);
                }
                CommandFound = 1;
        }
        if (strstr(cmd, "urn:") != NULL && strstr(cmd, ":service:") != NULL) {
                if ((TempPtr = strstr(cmd, "urn")) != NULL) {
                        memset(Evt->ServiceType, 0, sizeof(Evt->ServiceType));
                        strncpy(Evt->ServiceType, TempPtr,
                                sizeof(Evt->ServiceType) - 1);
                        CommandFound = 1;
                }
        }
        if (strstr(cmd, "urn:") != NULL && strstr(cmd, ":device:") != NULL) {
                if ((TempPtr = strstr(cmd, "urn")) != NULL) {
                        memset(Evt->DeviceType, 0, sizeof(Evt->DeviceType));
                        strncpy(Evt->DeviceType, TempPtr,
                                sizeof(Evt->DeviceType) - 1);
                        CommandFound = 1;
                }
        }
        if ((TempPtr = strstr(cmd, "::upnp:rootdevice")) != NULL) {
                /* Everything before "::upnp::rootdevice" is the UDN. */
                if (TempPtr != cmd) {
                        n = (size_t)TempPtr - (size_t)cmd;
                        strncpy(Evt->UDN, cmd, n);
                        Evt->UDN[n] = 0;
                        CommandFound = 1;
                }
        }
        if (CommandFound == 0)
                return -1;

        return 0;
}
```

CVE-2012-5960

We see that the code has been refactored, but still suffers from three buffer overflows in roughly the same places as before. The *strncpy()* function is being passed a length based on the distance between two strings in the attacker-supplied request, but is not checked against the size of the destination buffer.

Triggering CVE-2012-5958 can be accomplished with a request like the following:

```
M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:uuid:schemas:device:AAAA[…]AAAA:anything
Man:"ssdp:discover"
MX:3
```

Exploitability

What makes these vulnerabilities particularly bad is the context in which this code is called. An attacker can send a single, potentially spoofed UDP packet to the internet IP address of the system running this code and corrupt the program stack. This library limits input to approximately 2,500 bytes, providing ample room to include malicious code.

There are some caveats. The HTTP parser will reject the request if certain conditions are not met. The string that the attacker sends for the "ST" header that is processed by the *unique_service_name()* function must run the gauntlet of *scanner_get_token()* code within *httpparser.c* to avoid returning an error. This function treats characters differently depending on range and state.

1. Bytes between 0x20 and 0x7e are generally accepted
2. Bytes above 0x7e must be part of a double quoted string
3. Bytes below 0x20 or exactly 0x7f must NOT be placed in a double quoted string

These conditions complicate exploitation, especially on non-x86 platforms with fixed-width instruction sets such as ARM, MIPS, and PPC, due to limited encoding options available. The use of reserved bit manipulation techniques may help on fixed-width instruction platforms.

On all modern Linux platforms, address space layout randomization (ASLR), the lack of the execute bit on certain pages, and NULL bytes within the image address may also complicate exploitation. Notable, however, is the line of code that writes a NULL byte to the "end" of the copied string in TempBuf:

```
TempBuf[n] = '\0';
```

Depending on compiler options, the value of local variable "n" may have been overwritten by the preceding *strncpy()* function. This allows the attacker to write a single NULL byte to the location of their choice. This was proven possible on the little-endian MIPS platform against a commercial Linux-based home router and allowed a return address within the hosting binary to be used, even though it had a NULL byte in the address.

Additionally, partial overwrite techniques may be used on little endian platforms, but doing so will prevent stack control needed for return-oriented programming (ROP) techniques. Finally, on some platforms, it may be possible to simply predict the heap location of the allocated request, allowing a return into data that doesn't get processed by the HTTP parser.

Vendor Response

Rapid7 worked with CERT/CC to coordinate the disclosure process. CERT/CC reached out to the libupnp project along with other regional CERT teams to notify affected vendors. Version 1.6.18 of libupnp has been released and corrects the vulnerabilities identified in this paper.

MiniUPnP

The second most prevalent UPnP implementation is MiniUPnPd, a complete UPnP solution that handles not just the network processing, but the actual firewall rule management and other SOAP services.

Simple Service Discovery Protocol Vulnerabilities

Two related issues were identified in the SSDP handler of MiniUPnP. These issues were fixed in version 1.4 but had not been documented as security problems. CVE-2013-0229 has been assigned to track these issues.

In version 1.0 of the MiniUPnPd source code, the SSDP parser code lives within *minissdp.c*. The *ProcessSSDPRequest()* function is listed below:

```
void
ProcessSSDPRequest(int s, unsigned short port)
{
  int n;
  char bufr[1500];
  socklen_t len_r;
  struct sockaddr_in sendername;
  int i, l;
  int lan_addr_index = 0;
  char * st = 0;
  int st_len = 0;
  len_r = sizeof(struct sockaddr_in);

  n = recvfrom(s, bufr, sizeof(bufr), 0,
               (struct sockaddr *)&sendername, &len_r);

 [ … ]

  else if(memcmp(bufr, "M-SEARCH", 8) == 0)
  {
    i = 0;
    while(i < n)
    {
      while(bufr[i] != '\r' || bufr[i+1] != '\n')
        i++;
      i += 2;
      if(strncasecmp(bufr+i, "st:", 3) == 0)
      {
        st = bufr+i+3;
        st_len = 0;
        while(*st == ' ' || *st == '\t') st++;
        while(st[st_len]!='\r' && st[st_len]!='\n') st_len++;
      }
    }
```

CVE-2013-0229

The code in question is designed to receive a packet of up to 1500 bytes long and scan through it line by line until it finds one starting with the prefix “ST:”. The problem here is two-fold:

1. The first highlighted while loop doesn't check for the end of the buffer. If the request ends without a terminating \r\n sequence, the loop will continue reading on past the end of the allocated buffer. Eventually it will either find a spurious \r\n or crash when it runs off the stack.

2. The second and third highlighted while loops have the same problem in that they fail to check for an index beyond the end of the buffer. The first of these loops is unlikely to be hit, but the second loop will continue extending the length of the found string until a \r or a \n is found.

When triggered, the first issue is most likely to just crash the daemon, resulting in a denial of service. The second issue can be used to create a ridiculously long "ST" string that can go beyond the end of the buffer. In reality, this would have no noticeable effect, as the "ST" string is squashed into a snprintf() that limits the entire buffer size to 512 bytes.

The actual impact of both sets of issues would be a potential denial of service.

The example below would trigger a crash in version 1.0 of the SSDP service included in MiniUPnP version 1.0.

```
M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:uuid:schemas:device:MX:3< no CRLF >
```

SOAP Handler Vulnerabilities

A remote stack buffer overflow was identified in the SOAPAction handler of the HTTP service included with MiniUPnP version 1.0. All versions of MiniUPnP newer than 1.0 are not affected, but keep in mind that 67% of all identified systems running MiniUPnP are still using a vulnerable version. CVE-2013-0230 has been assigned to track this issue.

The following code is from *ProcessHTTPPOST_upnphttp()* in *upnphttp.c*.

```
    if(h->req_soapAction)
    {
      /* we can process the request */
      syslog(LOG_INFO, "SOAPAction: %.*s",
             h->req_soapActionLen, h->req_soapAction);
      ExecuteSoapAction(h,
        h->req_soapAction,
        h->req_soapActionLen);
    }
```

The ExecuteSoapAction method consists of the following code:

```
void
```

```
ExecuteSoapAction(struct upnphttp * h, const char * action, int n)
{
  char * p;
  char method[2048];
  int i, len, methodlen;

  i = 0;
  p = strchr(action, '#');
  methodlen = strchr(p, '"') - p - 1;

  if(p)
  {
    p++;
    while(soapMethods[i].methodName)
    {
      len = strlen(soapMethods[i].methodName);
      if(strncmp(p, soapMethods[i].methodName, len) == 0)
      {
        soapMethods[i].methodImpl(h);
        return;
      }
      i++;
    }

    memset(method, 0, 2048);
    memcpy(method, p, methodlen);
    syslog(LOG_NOTICE, "SoapMethod: Unknown: %s", method);
  }

  SoapError(h, 401, "Invalid Action");
}
```

CVE-2013-0230

1. The first two highlighted lines assume that the SOAPAction header contains a hash (#) and a double-quote. A missing hash would result in a NULL pointer read, crashing the service. A missing double-quote would result in a negative value for methodlen, since the *strchr()* would return NULL.

2. The memcpy() could be corrupted in two different ways.

    a. The first is triggered when a double-quote is missing from the SOAPAction header. This would result in the memcpy() being passed a negative length, which would be treated as a large positive length after being casted to an unsigned integer. This type of scenario is difficult to exploit unless the pointer to a signal handler is stored in adjacent memory to the destination pointer.

    b. The straightforward approach is to pass a quoted method length that exceeds 2048 bytes. This would result in a classic stack buffer overflow that has few character restrictions and nearly unlimited data to work with.

Vendor Response

Thomas Bernard of the MiniUPnP project was contacted with our findings and provided great feedback on an early draft of this advisory. The two sets of issues above were fixed in 2009

and 2008 respectively. In addition to the issues above, a few other issues have been identified and reported. This paper will be updated once a patch is available from MiniUPnP.

Please note that although the issues above have been fixed in newer versions of MiniUPnP, version 1.0 still accounts for 69% of the unique MinUPnP SSDP fingerprints and over 14% of the total UPnP fingerprints collected.

RAPID7

Appendices

Prior Research

The UPnP protocol has been the focal point of numerous security research efforts in the past. This section contains a listing of notable prior work in this area. This list is by no means comprehensive, but should provide a rough timeline of UPnP search research.

Microsoft Windows ME UPnP Service Denial of Service Flaw

In 2001, milo omega reported a denial of service flaw in the UPnP service included with Microsoft Windows ME.

Microsoft Windows UPnP Service Denial of Service Flaws

In 2001, Ken from FTU Security identified three vulnerabilities in the UPnP service included with Microsoft Windows.

Microsoft Windows UPnP Remote Code Execution Flaws

In 2001, researchers at eEye identified multiple exploitable vulnerabilities in the UPnP service included in Microsoft Windows.

Linksys Router Unauthorized Management Access Vulnerability

In 2002, Seth Bromberger reported an authentication bypass flaw in the Linksys web interface. The root cause was a reuse of the same interface for administration and UPnP SOAP requests, which lead to improperly handling of requests with a path ending in ".xml"

Vulnerability Report for Linksys Devices

In 2003, Gerard Richarte reported a number of flaws in the Linksys web interface, some of which were related to UPnP support.

NetGear FM114P WAN Information Disclosure

In 2003 Björn Stickler discovered that NetGear FM114P routers exposed the SOAP API on the WAN interface. This API could be used to retrieve information about the ISP connection, including the PPP username.

Xavi DSL Router UPNP Long Request Denial Of Service Vulnerability

In 2003, David F. Madrid reported a flaw in the Xavi DSL router UPnP HTTP service.

Belkin 54G Wireless Router Multiple Vulnerabilities

In 2004, pureone reported a number of vulnerabilities in the Belkin 54G wireless router, including an information disclosure issue in the UPnP service.

Shorewall and UPnP

In 2005, Thomas M. Eastep published a document entitled "Shorewall and UPnP", which details some of the weaknesses with the IGD profile of the UPnP protocol. The document contains the quote "From a security architecture viewpoint, UPnP is a disaster".

Microsoft Security Bulletin MS05-039

In 2005, Microsoft disclosed a remotely exploitable vulnerability in the Windows UPnP service. This vulnerability was reported by Neel Mehta and subsequently used in the Zotob worm.

Crazy Toaster: Can home devices turn against us?

In 2005, Dror Shalel presented at Defcon 15 on the UPnP protocol, with a focus on malicious device emulation.

D-Link Router UPNP Stack Overflow

In 2006, Barnaby Jack reported a remotely exploitable stack overflow in the UPnP SSDP service used by D-Link routers.

UPnP-Hacks.org Research

In 2006, Armijn Hemel created UPnP-Hacks.org, a web site detailing his investigation into UPnP security flaws. Issues identified include port forwarding, command injection through SOAP requests, and potential buffer overflows.

Microsoft Security Bulletin MS07-019

In 2007, Microsoft disclosed a remotely exploitable vulnerability in the Windows UPnP service. This vulnerability was reported by Greg MacManus.

Apple Mac OS X mDNSResponder Buffer Overflow

In 2007, Michael Lynn reported a remotely exploitable vulnerability in the Mac OS X mDNSResponder service. The flaw was triggered by an excessively long Location header in the UPnP SSDP response parser.

GNU Citizen/PDP UPnP Research

In 2008, GNU Citizen/PDP wrote three blog posts detailing some of the security issues with UPnP as a protocol. These were followed by an FAQ on the ability to report maps via IGD using a Flash application within the victim's web browser.

A Fox in the Hen House

In 2008, Jonathan Squire presented at BlackHat USA on security issues with the UPnP protocol and the IGD profile in particular. Links to the UPnPWn tool and associated white papers can be found on his blog.

Miranda UPnP Client

In 2008, SourceSec released the Miranda UPnP administration tool. This provided an easy way to identify UPnP control points and manipulate port forwarding rules. The latest version of this tool can be found in the Google Code repository.

UPnP Mapping at Defcon 19

In 2011, Daniel Garcia presented at Defcon 19 and described the widespread exposure of IGD SOAP interfaces to the public internet. Daniel also released "UMap", a tool for proxying attacks from the public internet to internal machines. CERT/CC issued VU#357851 as a result of this research.

Black Ops of TCP/IP 2011

In 2011, Dan Kaminsky also presented at Defcon 19 (as well as BlackHat USA 2011) in a talk that touched on UPnP. Starting on slide 31 is a brief discussion of the UPnP protocol. Dan's work focused on identifying internet-exposed SOAP interfaces bound to TCP port 2869 and referenced the work done by Daniel Garcia.

Vulnerable Products: Portable SDK for UPnP Devices (libupnp)

The list below was obtained by querying the SOAP device description of approximately one million UPnP-enabled devices. The vendor, product, and versions may not be accurate as a result of this method. Due to size constraints, the full list of vulnerable products has been placed online. Please follow this link to download the full list.

Vulnerable Products: MiniUPnP

The list below was obtained by querying the SOAP device description of approximately one million UPnP-enabled devices. The vendor, product, and versions may not be accurate as a result of this method. Due to size constraints, the full list of vulnerable products has been placed online. Please follow this link to download the full list.

Vulnerable Products: SOAP API Exposure

The list below was obtained by querying the SOAP device description of approximately one million UPnP-enabled devices. The vendor, product, and versions may not be accurate as a result of this method. Due to size constraints, the full list of vulnerable products has been placed online. Please follow this link to download the full list.