

Plug-N-Play Network Hacking

Universal Plug-N-Play (UPnP) is a protocol that allows various network devices to auto-configure themselves. One of the most common uses of this protocol is to allow devices or programs to open up ports on your home router in order to communicate properly with the outside world (Xbox, for example, does this). The UPnP protocol is built on top of pre-existing protocols and specifications, most notably, UDP, SSDP, SOAP and XML.

This article will address some of the security issues related to UPnP, briefly describe the inner workings of the protocol, and show how to identify and analyze UPnP devices on a network using open source tools. While we will be specifically focusing on IGDs (Internet Gateway Devices, aka, routers), it is important to remember that there are many other devices and systems that support UPnP as well, and they may be vulnerable to similar attacks.

del.icio.us

Discuss in Forums {mos_smf_discuss:heffner}

ChicagoCon 2009s Scheduled Speaker

Ethical Hacking Conference May 8 - 9

The Problem With UPnP

Allowing legitimate programs to alter your router settings as they need makes Joe User's life much easier - unfortunately, it also makes Joe Hacker's life easier, too. In order for UPnP to be truly "plug-n-play", there is no authentication built into the protocol; any program can use UPnP to alter a router's (or other UPnP device's) settings.

At the beginning of this year, PDP (of GNUCITIZEN fame) published research he had performed regarding the security of UPnP. While security problems with UPnP are nothing new, it is a protocol that is normally only run inside a LAN, forcing an attacker to infiltrate the network before exploiting any UPnP vulnerabilities. However, PDP showed that it was possible to use Flash to send UPnP requests from within a client's browser to a UPnP enabled router and change the router's firewall settings. Given that the Flash file can be embedded inside a malicious page, or injected into a trusted page via XSS or SQL injection, an attacker could leverage this to remotely alter a router's configuration. What's worse is that in most cases, these configuration changes are not reflected in the router's administrative interface, leaving the victim completely un-aware that anything has happened.

Of course, for wireless networks UPnP is dangerous even without the Flash attack, as an attacker can maintain physical distance from the network while still gaining access to the internal network. Once inside, he can begin altering the router's configuration settings via UPnP.

How Serious is This?

The seriousness of this attack depends on the UPnP implementation used by the router, as well as the router's configuration. For example, the configuration-altering Flash attack described above has to guess what a router's IP address is; this is usually fairly predictable, but some routers also assign themselves their own host names which takes any IP address guess work out of the picture. However, even if a router does not resolve to a host name, security through obscurity is not good practice, and simply changing a router's default IP address should not be considered a sufficient defense against such an attack.

In addition, some UPnP implementations are at more risk than others; several UPnP-enabled IGDs allow the DNS settings to be directly altered via UPnP, which is just begging for a MITM/phishing attack. Still others, particularly those devices that use some form of embedded Linux, actually use the un-sanitized values they receive via UPnP requests as part of executed shell commands, leaving them open to command-injection attacks. A list of some known vulnerable routers can be found [here](#).

OK, so let's say that no host name resolves to the router IP, and its UPnP implementation is not vulnerable to DNS-hijacking or command injection. Unfortunately, we're not out of the woods yet; remember that the most common use of UPnP is to open up ports on a router. Because of this, almost all routers that support UPnP also support the `AddPortMapping` action; this essentially allows a device or piece of software to tell a router "forward traffic coming to port X on the WAN to internal host Y who is listening on port Z". Well what if we forward traffic from port 8080 on the WAN to yahoo.com on port 80? I know, I know, yahoo.com isn't an internal host! But some UPnP implementations don't verify that they are forwarding to an actual internal host, thus allowing an attacker to bounce his traffic through the router. Further, if the attacker knows the router's internal IP or host name, he may be able to forward any port on the WAN side of the router to port 80 on the LAN side of the router, effectively opening up the router's administrative interface to the world.

An Overview of the UPnP Protocol

The UPnP protocol uses a multicast address of 239.255.255.250 and TCP port 1900. Devices that offer UPnP services will periodically send out SSDP NOTIFY messages to 239:255:255:250:1900, announcing themselves to any UPnP clients that are listening. If you watch the traffic on your LAN with a UPnP router attached, you will likely notice that it

sends out a burst of SSDP NOTIFY messages every few seconds; this is because most routers actually advertise themselves as multiple UPnP devices, and send one notification per device type.

Likewise, UPnP clients can send SSDP M-SEARCH requests to 239:255:255:250:1900 to see if any UPnP devices respond. Clients can send an M-SEARCH request looking for any UPnP device, or they can specify that they are looking for a particular UPnP device, or they can query only for a device that supports a specific UPnP service. UPnP hosts that match the devices/services requested will respond with a SSDP RESPONSE message, which contains the same information sent in an SSDP NOTIFY message.

An SSDP NOTIFY message sent by a UPnP host contains a 'Location' header which specifies the location of an XML file. This file contains XML data indicating, among other things, the UPnP device type(s) and services supported by the host, as well as paths to additional XML documents that describe the various services in detail. In order to discover the full UPnP capabilities of an IGD, you must parse through all the XML files to extract the devices types, services, and actions offered by the IGD.

UPnP devices support various services that advertise what actions they support. A UPnP client can send a UPnP device an action request at any time; this could be a request to open up a port, change the default DNS server, or anything else that the UPnP device supports. Input/output data for action requests and responses are sent using SOAP, which uses XML to structure information sent between two parties. SOAP requests are essentially HTTP POST requests with some additional SOAP headers included in the HTTP headers.

Auditing UPnP Devices Manually

To discover if a router supports UPnP, you can go to the router's administrative interface and check to see if there is an option to enable/disable UPnP; while most routers have UPnP enabled by default, some do not.

To actually audit a device's UPnP configuration, you can fire up Wireshark and look for SSDP NOTIFY packets being sent to the multicast address of 239.255.255.250 on port 1900; these notifications will be UPnP devices announcing themselves to the network. Once the SSDP NOTIFY messages are captured, you can examine Wireshark's dump of the SSDP headers to obtain the location of the root XML file. Once you retrieve that file (simply via an HTTP GET request), you can examine it to see what devices and services the UPnP host supports. Then, you can request additional XML files from the host (one file for each service) and parse those XML documents to determine what actions each service supports, and then correlate any related state variables (input/output variables) used for each action, as well as identifying which variables are used for input, and which are used for output (one action may use one variable as an input parameter, while another uses the same variable as an output parameter).

Obviously, auditing UPnP hosts manually can become extremely time consuming and forces us to manually generate SSDP requests to UPnP devices in order to run UPnP actions against them. Using a tool to automate the process makes life much easier.

Miranda is a UPnP administration tool written in Python. It sports its own command shell complete with tab-completion and command history, and provides you with the ability to save your work to a file that can be re-loaded for later analysis. You can also alter application settings on the fly, and log all of your commands to a log file, so you know exactly what you ran, and when.

Miranda can discover UPnP hosts either passively or actively, and all of a host's reported devices types, services, actions and variables can be enumerated with a single command. Service state variables are automatically correlated with their associated actions, and identified as either input or output variables for each action. Miranda stores all host information for all hosts in a single data structure, and allows you to directly traverse that data structure and view all of its contents.

Finally, you can run any action supported by a UPnP host; if that action requires input, you will be told the input name and type (string, 4 byte integer, 2 byte integer, etc), as well as any allowed values or value ranges that the UPnP host has supplied, and prompted to input a value.

Discovering UPnP Hosts With Miranda

When you first start Miranda, it will drop you into an interactive shell with a 'upnp>' prompt. The first thing you will likely want to do is discover if there are any UPnP hosts on your network; this can be done with the 'pcap' or 'msearch' commands. When given the 'pcap' command, Miranda will passively listen for SSDP NOTIFY messages, while the 'msearch' command will query for UPnP devices using an M-SEARCH message. By default, 'msearch' will look for any/all UPnP devices, but you can also tell it to search for a specific device type or service if you wish. In this example, we'll just look for any devices:

```
upnp> msearch
```

```
Entering discovery mode for 'upnp:rootdevice', Ctrl+C to stop...
```

```
*****
```

```
SSDP reply message from 192.168.0.1:5678
```

```
XML file is located at http://192.168.0.1:5678/igd.xml
```

```
Device is running Embedded UPnP/1.0
```

Discover mode halted...

Here we can see that there is one UPnP host on the network, which happens to be my D-Link DI-524 router. We can also see its reported UPnP server type ('Embedded UPnP/1.0'), and the location of its root UPnP XML file.

Running the 'host list' command shows a list of all discovered UPnP hosts and each host's index number (the index number is used in subsequent commands to reference a specific host):

```
upnp> host list
```

```
[0] 192.168.0.1:5678
```

We have discovered a UPnP host, now we need to enumerate its capabilities. Running the 'host get 0' command will get all of the UPnP information advertised by the host with the index number of 0:

```
upnp> host get 0
```

Requesting device and service info for 192.168.0.1:5678 (this could take a few seconds)...

Host data enumeration complete!

Now we can look at all the data we have collected for this host using the 'host info' command. This command allows us to traverse Miranda's internal host data structure and view all of the information that it stores. Note also that all of these commands will tab-complete so you don't have to type them all out:

```
upnp> host info 0
```

```
xmlFile : http://192.168.0.1:5678/igd.xml
```

```
name : 192.168.0.1:5678
```

```
proto : http://
```

```
serverType : Embedded HTTP Server 3.23
```

```
upnpServer : Embedded UPnP/1.0
```

```
dataComplete : True
```

```
deviceList : {}
```

```
upnp> host info 0 deviceList
```

InternetGatewayDevice : {}

WANDevice : {}

WANConnectionDevice : {}

upnp> host info 0 deviceList WANConnectionDevice services WANIPConnection actions

AddPortMapping : {}

GetNATRSIPStatus : {}

GetGenericPortMappingEntry : {}

GetSpecificPortMappingEntry : {}

ForceTermination : {}

GetExternalIPAddress : {}

GetConnectionTypeInfo : {}

GetStatusInfo : {}

SetConnectionType : {}

DeletePortMapping : {}

RequestConnection : {}

Depending on the host, there can be a lot of data in this structure, so if you just want to see a summary of the data for a particular host, run the 'host summary' command:

upnp> host summary 0

Host: 192.168.0.1:5678

XML File: <http://192.168.0.1:5678/igd.xml>

InternetGatewayDevice

manufacturerURL: <http://www.dlink.com>

modelName: D-Link Router

UPC: 123456789001

modelName: None

presentationURL: <http://192.168.0.1:80>

friendlyName: D-Link Router

fullName: urn:schemas-upnp-org:device:InternetGatewayDevice:1

modelDescription: Internet Access Router

UDN: uuid:upnp-InternetGatewayDevice-1_0-12345678900001

modelURL: None

manufacturer: D-Link

WANDevice

manufacturerURL: <http://www.dlink.com>

modelName: D-Link Router

UPC: 123456789001

modelName: 1

presentationURL: None

friendlyName: WANDevice

fullName: urn:schemas-upnp-org:device:WANDevice:1

modelDescription: Internet Access Router

UDN: uuid:upnp-WANDevice-1_0-12345678900001

modelURL: <http://support.dlink.com>

manufacturer: D-Link

WANConnectionDevice

manufacturerURL: <http://www.dlink.com>
modelName: D-Link Router
UPC: 123456789001
modelName: 1
presentationURL: None
friendlyName: WAN Connection Device
fullName: urn:schemas-upnp-org:device:WANConnectionDevice:1
modelDescription: Internet Access Router
UDN: uuid:upnp-WANConnectionDevice-1_0-12345678900001
modelURL: <http://support.dlink.com>
manufacturer: D-Link

The summary command shows us all of the device types that this host is reporting, as well as some additional data related to each device type. If you want to view all of the information that Miranda has stored about a particular host, you can do so with the 'host details 0' command. However, the output from this command is often quite lengthy, so you will probably want to save it to file and view it in a text editor instead; this can be done with the 'save info 0' command:

```
upnp> save info 0 dl524
```

Host info for '192.168.0.1:5678' saved to 'info_dl524.mir'

The 'dl524' argument is optional; if no string is supplied here, then the host index number will be used in its place. While we're at it, we can also save all of the data that Miranda has stored about all of the hosts that have been discovered so that we can later load it back into Miranda during a new session:

```
upnp> save data session1
```

```
Host data saved to 'struct_session1.mir'
```

Sending UPnP Commands With Miranda

Now let's explore some of the actions we can run on this device. Earlier while running the 'host info' command, we listed all of the actions available for the WANIPConnection service that is associated with the WANConnectionDevice device; let's look at those again:

```
upnp> host info 0 deviceList WANConnectionDevice services WANIPConnection actions
```

```
AddPortMapping : {}
```

```
GetNATRSIPStatus : {}
```

```
GetGenericPortMappingEntry : {}
```

```
GetSpecificPortMappingEntry : {}
```

```
ForceTermination : {}
```

```
GetExternalIPAddress : {}
```

```
GetConnectionTypeInfo : {}
```

```
GetStatusInfo : {}
```

```
SetConnectionType : {}
```

```
DeletePortMapping : {}
```

```
RequestConnection : {}
```

As you can see, this is the service that supports the AddPortMapping and DeletePortMapping actions, as well as a few other interesting looking ones. First we'll try running the GetExternalIPAddress action; this can be done with the 'host send' command. To run a command, you must supply the host index number, the name of the device type that supports the service, the name of the service that supports the action, and the name of the action you want to run (again, all of these fields tab-complete, so it's not as much typing as it looks):

```
upnp> host send 0 WANConnectionDevice WANIPConnection GetExternalIPAddress
```

```
NewExternalIPAddress : 68.12.34.56
```

'NewExternalIPAddress' is the name of the variable associated with this action, and '68.12.34.56' is the value returned for that variable by the IGD. Some actions have several variables associated with them; these variables can be either input (values we have to supply to the IGD) or output (values returned back to us by the IGD). In the case of the GetExternalIPAddress action, there is only one output variable. If, however, there are input variables available for an action, Miranda will prompt us for those values before sending the action. All variable information can be enumerated/viewed using the 'host info' or 'host details' commands, but we do not have to have any knowledge of these variables before we run UPnP actions.

Let's try adding a port mapping to open up the administrative interface on port 80 of the IGD (192.168.0.1) by mapping it to port 8080 on the WAN:

```
upnp> host send 0 WANConnectionDevice WANIPConnection AddPortMapping
```

Required argument:

Argument Name: NewPortMappingDescription

Data Type: string

Allowed Values: []

Set NewPortMappingDescription value to: All your ports are belong to us

Required argument:

Argument Name: NewLeaseDuration

Data Type: ui4

Allowed Values: []

Set NewLeaseDuration value to: 0

Required argument:

Argument Name: NewInternalClient

Data Type: string

Allowed Values: []

Set NewInternalClient value to: 192.168.0.1

Required argument:

Argument Name: NewEnabled

Data Type: boolean

Allowed Values: []

Set NewEnabled value to: 1

Required argument:

Argument Name: NewExternalPort

Data Type: ui2

Allowed Values: []

Set NewExternalPort value to: 8080

Required argument:

Argument Name: NewRemoteHost

Data Type: string

Allowed Values: []

Set NewRemoteHost value to:

Required argument:

Argument Name: NewProtocol

Data Type: string

Allowed Values: ['TCP', 'UDP']

Set NewProtocol value to: TCP

Required argument:

Argument Name: NewInternalPort

Data Type: ui2

Allowed Values: []

Set NewInternalPort value to: 80

The AddPortMapping action takes several input values, but there are a few important notes that should be made here:

- 1) Boolean values are either '1' (true) or '0' (false);
- 2) The NewProtocol argument only allows for two values, 'TCP' or 'UDP'
- 3) We did not specify a value for the NewRemoteHost variable, which allows all remote hosts to match this port mapping.

We didn't get any output from the AddPortMapping action because it has no output variables defined (again, this information is all stored in the data structure if you're curious). However, we can verify that the action was successful by running the GetSpecificPortMappingEntry action:

```
upnp> host send 0 WANConnectionDevice WANIPConnection GetSpecificPortMappingEntry
```

Required argument:

Argument Name: NewExternalPort

Data Type: ui2

Allowed Values: []

Set NewExternalPort value to: 8080

Required argument:

Argument Name: NewRemoteHost

Data Type: string

Allowed Values: []

Set NewRemoteHost value to:

Required argument:

Argument Name: NewProtocol

Data Type: string

Allowed Values: ['TCP', 'UDP']

Set NewProtocol value to: TCP

NewPortMappingDescription : All your ports are belong to us

NewLeaseDuration : 0

NewInternalClient : 192.168.0.1

NewEnabled : 1

NewInternalPort : 80

Now we can delete the port mapping with the DeletePortMapping action. Like AddPortMapping, the DeletePortMapping action provides no output unless an error occurs:

```
upnp> host send 0 WANConnectionDevice WANIPConnection DeletePortMapping
```

Required argument:

Argument Name: NewProtocol

Data Type: string

Allowed Values: ['TCP', 'UDP']

Set NewProtocol value to: TCP

Required argument:

Argument Name: NewExternalPort

Data Type: ui2

Allowed Values: []

Set NewExternalPort value to: 8080

Required argument:

Argument Name: NewRemoteHost

Data Type: string

Allowed Values: []

Set NewRemoteHost value to:

There are plenty of other interesting actions, such as ForceTermination, which causes the router to drop its WAN connection, so it is worthwhile to explore the various services and actions for each device you are auditing.

Conclusion

While UPnP is a protocol that few understand, it is active on the vast majority of home networks, and even on some corporate networks as well. Many devices support UPnP in order to ease the use for consumers, however, they often support actions that no service should be allowed to perform automatically, and especially not without authorization. Worse, the protocol implementation itself is rarely built with a security mindset, leaving it open to further exploitation.

The best defense against local/remote exploitation of UPnP is to simply disable the feature on any/all network devices. However, considering that this protocol and other "auto-magic" protocols are designed to assist Joe User, who likely is unaware of the dangers of such protocols, the only true solution is for vendors to become more attentive to their designs and more secure in their implementations.

[Click for 893x541 Version of Cracked UPnP Logo](#)