University of Amsterdam

System and Network Engineering

Research Project 1

Universal Plug and Play vulnerabilities in Eventing

Author:
Joeri Blokhuis
joeri.blokhuis@os3.nl

Supervisor:
Armijn Hemel

February 4, 2009

Contents

Abstract

UPnP technology is used to auto-configure networked devices and to simplify home networking. The eventing part of UPnP keeps state of variable changes. A UPnP control point can submit a subscription to be notified of event changes. The device will notify the control point when state changes occur. To receive a notification a Callback URL is submitted when subscribing to the eventing service. This URL is not restricted to the subscribers' IP address. Instead any IP address can be registered. When testing devices a flaw was found in subscribing to event notifications, causing a denial of service.

1 Introduction

Configuring a networked device properly often requires some knowledge of networking, which many people do not have. To overcome the problems resulting from improper configuration (not properly working networked devices) technologies such as Universal Plug and Play (UPnP)[1], JINI[2] and Zeroconf[3] were developed.

Home networking devices such as game consoles, digital picture frames and Internet Gateways, is a rapidly growing market. Therefore in 1999 the UPnP Forum[4] was formed by twenty major companies from various industries, like, computers, home automation, networking, consumer electronics and mobile products. Ten years later more than 800 companies are members of the UPnP Forum. The UPnP Forum is an organization to help define the UPnP standards and to simplify home networking.

UPnP technology enables data communication between two devices which is controlled by a so called control device. UPnP is also operating system, programming language or network independent. When two devices are communicating, configuration is done transparently on the background, no user interaction is required. For instance, a MSN video chat requires a port to be opened, to set up a peer-to-peer communication. With UPnP enabled this will be done automatically without any user interaction. When the video chat has finished, the port will be automatically closed.

UPnP is using industry standards including TCP, IP, UDP, HTTP, SOAP and XML. A device can join a network, obtain an IP-Address, learn about other devices and leave the network without leaving any unwanted state.

UPnP has a simple architecture. The device architecture can be defined by devices, services and control points. A device is called a UPnP device when it implements the protocols required by the UPnP architecture. A service is no more than the functionalities provided by a device. A control point is an entity that works with services of a UPnP device. A control point can invoke a service, this can be a desktop PC or another UPnP device. In this way they can form a peer-to-peer network and take advantage of each-other services. A service might have a state table. A control point can subscribe to that service to keep state when a variable changes. This is called eventing.

UPnP devices have several phases and will be described briefly.

Addressing A UPnP device joins a network and obtains an IP-Address.

Description Lists the functionality provided by a device. Descriptions are written in XML-format.

Discovery Control points find a UPnP device and retrieve description information.

Control Allows a control point to invoke actions.

Eventing A device will notify all subscribers when state changes occur.

Presentation Optionally a device will provide an HTML-based interface for monitoring and alternating.

Because of the time frame of four weeks, it is infeasible to research all phases. Therefore, only UPnP Eventing will be researched. The main question will be:

What vulnerabilities can be found in eventing? This will be done by analysing code and testing devices.

In section 2 a brief overview is given about other researches conducted on Universal Plug and Play. Section 3 will outline UPnP eventing in detail by explaining how to register and cancel for event notification messages. An approach to tackle UPnP eventing is discussed in section 4. Finally section 5 will present a conclusion and section 6 discusses future work.

2 Background

Armijn Hemel showed that UPnP is prone to various attacks. His research[8] focussed on Internet Gateway Devices(IGD) and showed that internal machines could be exposed to outside networks. The internal machine does not even have to know that a port on his IP-address is forwarded to the outside network. Another vulnerability he had found was, invalidated parameters. By making a malicious port forwarding call it was possible to reboot the device.

Adrian Pastor and Petko D. Petkov showed[9] that it was possible to abuse a known defect in a Flash player to send UPnP requests from within a client's web browser to a UPnP enabled router and change the router's firewall settings.

Dror Shalev created a "crazy toaster". Shalev wants to make people aware of what danger UPnP devices can be, by creating Trojan UPnP devices[10], that will provide services that could do serious harm.

Jonathan Squire confirmed the previous mentioned work in his presentation[11] and developed a tool called UPnPwn. This tool allows basic manipulation off an IGD, such as adding, removing and listing port mappings.

3 Eventing

As mentioned before, eventing allows control points to register to a device when state changes occur. Control points receive a notification and may wish to respond to that notification, by invoking an action.

UPnP is using a publisher/subscriber model to implement eventing. The General Event Notification Architecture (GENA) is used to implement this. A publisher is a service which accept registrations from clients who are interested in receiving notifications. A subscriber is called a ”client” and will subscribe itself to a publisher. If a subscriber does not wish to receive any event notifications it may unsubscribe itself from the publisher. The publisher will then, stop sending event messages on state changes. Subscriptions may expire overtime. In order to keep receiving message notifications, a renewal subscription must be submitted by the client.

3.1 GENA

The GENA protocol is used to implement the publisher/subscriber model. GENA uses HTTP as the transport layer for communication between publishers and subscribers. To achieve this GENA has introduced three new HTTP methods for subscription and notification:

- SUBSCRIBE to register for events and receive notifications. This is also used for renewing existing subscriptions.
- UNSUBSCRIBE to cancel subscriptions.
- NOTIFY to send events to the subscriber.

Subscribe and unsubscribe messages are always sent by a control point and never from a UPnP device, having an event service. A UPnP device that provides eventing will receive subscribe and unsubscribe messages and will only send notify messages to control points.

GENA messages use four new HTTP headers along with common headers such as Host, Timeout, Date, Server, Content-Length and Content-Type. The new headers are:

- CALLBACK is a URL to send notifications to the subscriber.
- NT provides the notification type. This is used to indicate the kind of notification.
- NTS notification subtype. NTS is used when a more detailed notification is selected.
- SID stands for Session ID. This ID is used when canceling or renewing a subscription, but also for communication between a subscriber/publisher.

3.2 Subscription

Before any subscription can be made a control point needs to retrieve a description from a UPnP device that supports eventing. This description is written in XML-format and contains information about subscribing to eventing. For eventing the following should be found: an event subscription URL and a service identifier for each service.

A subscription request will be of the following format:

```
SUBSCRIBE: publisher path HTTP/1.1
HOST: publisher HOST:PORT
CALLBACK: delivery URL
NT: upnp:event
Timeout: Second-subscription duration
```

Table 1 explains the subscription headers.

HTTP Header	Type	Example
Host	domain or IP address and optional port number (of the service)	192.168.2.1:52896
Callback	One or more URLs to deliver events	<http://192.168.2.1/test> <nextURL>
NT	upnp:event	Must always be upnp:event
Timeout	Second-time (in seconds or Infinite)	Second-1800 or Seconds-Infinite

Table 1: Subscription headers.

A subscription is sent without a body, a CR/LF (carriage return/line feed) should be entered after the last HTTP header. Of all headers, only Timeout is not required. However it is recommended that a timeout should be added to a subscription by the control point. If no timeout is given a default value will be set by the publisher.

3.2.1 Subscription response

When a subscription is received by a device, it will accept the subscription when the supplied headers were parsed successfully. The service will respond with a unique Session ID (SID) and the duration of the subscription. The SID provides uniqueness and allows further communication between the service and control point.

When the subscription is successful the service will send a response in the following format:

```
HTTP/1.1 200 OK
DATE: date of when the response was generated
SERVER: Operating System/version UPnP/1.0 product/version
SID: uuid:subscription-UUID
Timeout: Second-subscription duration
```

The response must be sent within 30 seconds.

Note: According to the UPnP specifications, the service should accept as many subscriptions as it can reasonably maintain and deliver.

When a subscription is not accepted by the publisher it should return an HTTP error. There are four error values that can occur:

Type	Response
Incompatible headers	400 Bad Request
Missing or Invalid Callback	412 Precondition Failed
Invalid NT	412 Precondition Failed
Unable to Accept Subscription	5xx

Table 2: HTTP error response.

3.2.2 Subscription renewal

The duration of a subscription is determined by a control point or by the service. When no timeout (in seconds) is submitted during a subscription by a control point, the publisher will set a default timeout. This will be typically 1800 or more seconds, depending on the vendor. When a subscription is not infinite the subscription will expire at some stage and will be removed from the publisher's subscribers list. However, if the control point wishes to receive messages after the subscription has expired it must renew the subscription before it expires.

The renewal subscription is sent to the same URL as the initial subscription and is actually the same as a subscription only using different HTTP headers. For renewal of a subscription the Callback and NT header are no longer required, because the publisher has already registered those headers during the initial subscription. A renewal subscription only adds the SID header, which was returned in the response of a successful subscription.

The format of a renewal subscription is as follows:

```
SUBSCRIBE: publisher path HTTP/1.1
HOST: publisher HOST:PORT
SID: uuid:subscription UUID
Timeout: Second-subscription duration
```

The unique Session ID is stored by both the publisher and control point for further communication. Therefore, only the SID is required to renew a subscription. The Timeout field is not required, but its use is recommended.

After the renewal request is accepted by the publisher it will respond with the same message format as the response to the original subscription request. This includes Date, Server, SID and Timeout headers. Unlike a subscription there will be no initial event message sent to the control point. This is not needed because the control point is already receiving event notifications on state changes.

If the renewal cannot be accepted by the publisher, it will return one of the previous mentioned HTTP errors (table 2).

3.3 Cancellation

There are two main reasons to cancel a subscription. The most prominent reason for a control point to cancel notification, is that it is no longer desired. The second reason is when the control point leaves the network. The control point must say proper goodbye to the publisher when leaving the network. If the control point does not send a cancellation message the publisher will keep the control point registered in the subscribers list. A control point will stay in the subscribers list until the subscription expires or is cancelled. Meanwhile, event messages are sent to the control point regardless of its presence. This will consume unnecessary bandwidth and processing power on the publisher's side. To avoid unnecessary consuming the control point is ought to cancel a subscription when leaving the network. This is done by sending an unsubscription message to the publisher. An unsubscription message has the following format:

```
UNSUBSCRIBE: publisher path HTTP/1.1
HOST: publisher HOST:PORT
SID: uuid: subscription UUID
```

All it takes to unsubscribe a control point from receiving event notifications is the SID, which is generated after a successful accepted subscription.

When a subscription is successfully received and processed, the service sends an HTTP success response to the control point. This is a typically:

```
HTTP/1.1 200 OK
```

If some error occurs with the cancellation request, the service will send a response with one of the following errors:

Error type	**Error return**
Incompatible headers	400 Bad Request
Invalid SID	412 Precondition Failed
Missing SID	412 Precondition Failed

Table 3: Cancellation error response.

3.4 Notification

As mentioned previously, a service will send an initial event message immediately after a successful subscription of a control point. This message contain all the names and values for all evented variables provided by the service in XML format. Control points are kept informed of state changes as soon as any of the evented variable changes state.

A notify message has the following format:

```
NOTIFY delivery path HTTP/1.1
HOST: delivery host:port
CONTENT-TYPE: text/xml
CONTENT-LENGTH: length of body in bytes
NT: upnp:event
NTS: upnp:propchange
SID: uuid:subscription UUID
SEQ: event key

<e:propertyset xmlns:e="urn:schemas-upnp-org:event-1-0">
....<e:property>
.......<variableName>new Value</variableName>
....</e:property>
</e:propertyset>
```

This message is sent by using the HTTP Notify method, which is introduced by GENA. When such a message is received, it will indicate that this is an event. When a notify message is received by a control point is must respond with an HTTP success.

```
HTTP/1.1 200 OK
```

If the subscriber does not respond within 30 seconds the service will stop trying to send the message. However the subscription remains active and will try to send future event messages (on next event) to the subscriber. The subscriber will not be deleted from the subscribers list until the subscription is cancelled or has expired.

3.4.1 Event keys

When a NOTIFY message is received it will contain a SEQ header. A SEQ header contains an event key(sequence number). This number is used as a control mechanism to ensure clients have not missed an event notification. The event key is a 32-bit integer value and is maintained by the publisher. The initial notification message is initialized on 0, this message is sent to the control point when the publisher successfully accepted the subscription request. On every subsequent message sent by the publisher, the event key will increment. When the event key gets too large it should be handled by the publisher and control point. The publisher is responsible for resetting the event key to 1. The event key will not be wrapped back to 0, because this represents the initial notification message. The control point however, must not interpret this as an error.

When a control point detects it has missed an event notification message, it must unsubscribe and re-subscribe itself. This is done to ensure that the control point has the current state variables. By re-subscribing, the control point will receive a new session ID and the initial notification message with a 0 event key.

4 Eventing in practice

In this section the practical part of eventing will be discussed. During this research two angles of attack were researched:

- For subscribing, a Callback URL will be registered. This URL is called by the publisher when a variable has changed. This will be researched whether it is possible to attack a website by changing the Callback URL into any given URL.
- If a subscriber leaves the network and does not cancel its subscription, the publisher will think the subscriber is still alive and will not be deleted from the subscribers list. By abrupt terminating from the network and making a new subscription when rejoining the network, will it be possible to fill up the subscribers list in such a way that it cannot accept any new subscribers?

4.1 Methodology

This subsection discusses the methodology used during this research.

4.1.1 Testing UPnP tools

Two Internet Gateway Devices with UPnP capabilities were used for testing:

- Model: Edimax BR-6104K[6]
  Firmware: 3.25
  UPnP library: Intel libupnp 1.2.1[13]
- Model: Sitecom WL-534[5]
  Firmware: 1.52
  UPnP library: Intel libupnp 1.2.1[13]

To get a better understanding on the practical side of UPnP, I used a tool Miranda[16] and a tool written by my supervisor Armijn Hemel to test and debug the devices. Both tools are written in Python and have not fully implemented all functionality which a normal UPnP device/control point might provide. However it contains basic functions, such as sending discovery messages, retrieving and parsing XML description files and adding/deleting portmappings. A TCP dump tool was used to analyse UPnP messages that were sent on the network. This gave a good overview of how UPnP works in practice on the two tested devices.

4.1.2 Code analysing/writing

Code archives of the Edimax[14] en Sitecom[15] which are publicly available have been investigated on potential flaws in UPnP Eventing. Both stacks use libupnp[13], a UPnP library written by Intel. It is written in C and although it is documented it is hard to understand the source code in the limited amount of time. A more experienced C-programmer might disagree.

Eventing was not implemented in both of the previous mentioned tools. Therefore a tool was written (in Python) to send subscribe, re-subscribe and unsubscribe messages.

4.2 Subscription

Before a subscription can be made by a control point, a description file must be retrieved to set the right HTTP headers such as HOST and publisher path. The description file is in XML format and contains information like manufacturer, model, presentation URL and services provided by the UPnP device. A description file can be retrieved by sending a discovery message on the network to the the multicast address 239.255.255.250 on port 1900 via UDP. A UPnP enabled device is required to respond with a message similar to following:

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://192.168.2.1:52869/picsdesc.xml
NT: upnp:rootdevice
NTS: ssdp:alive
SERVER: Linux/2.4.18-MIPS-01.00, UPnP/1.0, Intel SDK for UPnP devices /1.2
USN: uuid:75802409-bccb-40e7-8e6c-fa095ecce13e::upnp:rootdevice
```

The LOCATION header from the message above contains a URL to an XML file. This file must be retrieved, to locate the eventSubURL. The following is an extract from the XML file and is only showing a part of the service list.

```
......<serviceList>
.......<service>
........<serviceType>urn:schemas-upnp-org:service:WANIPConnection:1</serviceType>
........<serviceId>urn:upnp-org:serviceId:WANIPConn1</serviceId>
........<controlURL>/upnp/control/WANIPConn1</controlURL>
........<eventSubURL>/upnp/control/WANIPConn1</eventSubURL>
........<SCPDURL>/picsconnSCPD.xml</SCPDURL>
.......</service>
......</serviceList>
```

The eventSubURL must be extracted, because it serves as the delivery path of a subscription message. The LOCATION header and eventSubURL are needed to create a proper subscription message. The LOCATION header provides information needed for the HOST header, http://192.168.2.1:52869 and the XML file contains the publisher path, /upnp/control/WANIPconn1. The following piece of Python code shows how a subscription message is created according to the extracted information.

```
headers = {'HOST': '', 'CALLBACK': '', 'NT': 'upnp:event', 'TIMEOUT': 'Second-infinite'}
#Host and optional port number
conn = httplib.HTTPConnection("192.168.2.1:52869")

#Assign dynamic headers with proper values
headers['HOST'] = "192.168.2.1:52869"
headers['CALLBACK'] = "<192.168.2.101/test>"

#create request. We want to subscribe so the HTTP method request is placed as the first ar
#and the second argument is the selector or the delivery path (like: /index.html)
```

```
conn.putrequest('SUBSCRIBE', "/upnp/control/WANIPConn1", skip_host=True, skip_accept_encod

# Put all headers in order to the specifications
conn.putheader("HOST", headers['HOST'])
conn.putheader("CALLBACK", headers['CALLBACK'])
conn.putheader("NT", headers['NT'])
conn.putheader("TIMEOUT", headers['TIMEOUT'])
conn.endheaders()

#wait for a response and print it to the commandline
response = conn.getresponse()
print response.status, response.reason
```

On the network the subscription and response will be as follows:

```
The subscription looks like this:

SUBSCRIBE /upnp/control/WANIPConn1 HTTP/1.1
HOST: 192.168.2.1:52869
CALLBACK: <http://192.168.2.101/test>
NT: upnp:event
TIMEOUT: Second-infinite

The response looks like this:

HTTP/1.1 200 OK
DATE: Sat, 01 Jan 2000 22:30:45 GMT
SERVER: Linux/2.4.18-MIPS-01.00, UPnP/1.0, Intel SDK for UPnP devices /1.2
SID: uuid:16766c80-1dd2-11b2-a2ce-e7182fbea8a1
TIMEOUT: Second-infinite
```

4.2.1 Callback URLs

The Callback URL is submitted manually by the control point. In the subscription example the IP address of the subscriber was used. While testing different Callback URLs, it was found that any URL could be submitted by the control point. This means that control point A could request a subscription for control point B or any other random website. The service will accept the URL as long as it is valid, regardless of the source of the IP address. The service will also accept non-local IP addresses. When an event occurs the publisher will send its notifications to the registered Callback URL. When just a regular website is submitted as Callback URL, the website will receive the notifications. Because a regular website most likely does not support an HTTP NOTIFY method, the website will respond with:

```
501 Method Not Implemented
```

The devices that were researched will not cancel the subscription as one might expect and as is required by the specification. It looks like the same principle is applied when a control point does not respond to notifications. Instead it will maintain subscriptions until they expire. By keeping faulty control points (which can be a mistaken IP address) subscribed, sending events will consume unnecessary resources. This contradicts with a previous statement (according to the specifications[12]) on cancelling event notifications. When leaving the network a cancellation message should be sent, because it will reduce the service and network load. It could not be found why subscriptions are kept when a 501 ERROR is received.

The service will also allow the same URL to be registered more than once. The service does not check its subscribers list for the same Callback URL, but instead it registers the same URL over and over again.

Subscription requests allow more than one Callback URL to be submitted. When submitting multiple URLs, they should be separated by angle brackets < >. During tests using multiple URLs it was not clear why state changes were only sent to one URL at time which also happened to be the first URL in the subscribers list. By doing some more thorough testing it became clear that the stack will check the URLs in order of subscription. When a "correct" URL is found it will be the only one used for the Callback. For example:

```
CALLBACK: <http://192.168.2.1/test><http://google.com/>
```

This example shows multiple callback URLs. When a subscription is received by the service it will process the first URL. Because <http://192.168.2.1/test> is a correct URL no further processing will be done. The service only checks whether the submitted URL is valid, but does not check if it accept NOTIFY messages.

Another example:

```
CALLBACK: <http://google.com/><http://192.168.2.1/test>
```

In this example the two URLs are switched. According to the UPnP specifications a domain should be accepted. However this is not the case. All domains are rejected by getting a 412 PRECONDITION FAILED response. Processing the second URL, which is correct, will be used as the Callback URL.

A successful subscription will reply among other HTTP headers with a unique generated Session ID. This will be discussed in the next section.

4.3 UUID

A successful subscription will have a HTTP header with a unique, generated Session ID. A UUID (Universally Unique IDentifier), also known as a GUID (Globally Unique IDentifier), is 128 bits long and is used for communication between the publisher and control point. In libupnp the UUID is generated using a timestamp and a clock sequence to create randomness. Once a UUID is generated it should be kept private to the subscriber/control point and the publisher.

Every event notification message sent to a certain control point will contain the same unique UUID and does not change during the subscription. For a re-subscription or cancellation the UUID is used to extend a subscription or unsubscribe from receiving events. The sender of these messages will not be checked. This means a cancellation could be sent from any local IP address.

This implementation has one advantage: when an IP address of a host suddenly changes, it can still send a proper cancellation message and subscribe itself again by setting the Callback URL to the new IP address. The only downside of this implementation is that a UUID can be sniffed by ARP-spoofing the network. If a NOTIFY message is captured a cancellation message is easily sent to the publisher.

4.4 Denial of Service

Although the initial plan was to attack the service its subscribers list by leaving and rejoining the network, a simpler way was found. The same Callback URL is allowed to be used for an unlimited amount of times and thereby it has become suitable for the same attack: Fill the subscribers list of the service in such a way that it cannot accept any new subscriptions. By setting the timeout to 'infinite' a subscribed message will never expire until UPnP is disabled.

This attack is done by creating a while-loop in which a new subscription request is sent over and over again. The Callback URL is set to the same IP address and contains a unique part, which is simply the iteration number of the while-loop. This means when the loop is at 421 the Callback URL will look something like: http://192.168.2.1/421. In this way output can be easily monitored on the control point. This approach was tested on both IGD devices and both were prone to this attack, which was to be expected since both stacks use libupnp. On the Sitecom device it took on average around 14000 subscriptions to create a denial of service. The Edimax device needed around 18000 subscriptions on average before it stopped working.

Before the mass subscription request the output of Nmap looked like this:

```
joeri@localhost#nmap -v -PN -p 52869 192.168.2.1
Interesting ports on 192.168.2.1:
PORT      STATE SERVICE
52869/tcp open  unknown
```

After the mass subscription request the output of Nmap looked like this:

```
joeri@localhost#nmap -v -PN -p 52869 192.168.2.1
Interesting ports on 192.168.2.1:
PORT      STATE SERVICE
52869/tcp closed unknown
```

It clearly shows that UPnP is no longer working.

While testing devices different values for the sleep commands were used in the while-loop. A sleep function is used to influence the amount of subscriptions being sent. The following two tables 4 and 5 show how long and how many subscriptions it took to cause a denial of service. It also shows the amount of milliseconds for the sleep function in the while-loop, because this has a noticeable effect on the Edimax device.

Sleep(ms)	**Subscriptions(average amount)**	**Time(minutes)**
0.2	14187	98
0.2	13567	94
0.15	13895	95

Table 4: Sitecom statistics

Sleep(ms)	**Subscriptions(average amount)**	**Time(minutes)**
0.2	23744	102
0.10	18567	46
0.0	17485	42

Table 5: Edimax statistics

When testing the Edimax router the program was set to sleep to 0.2 seconds on every loop. It took around 23000 subscriptions and about 1,5 hour on average to cause a denial of service. When choosing a lower timing on the sleep function it took less subscriptions to stop UPnP from working. Eventually with no sleep function it took around 18000 subscriptions and only 42 minutes on average. On the Sitecom router a sleep function had to be used otherwise it would receive a timeout response from the service. Using different sleep values did not have much influence on the average of subscriptions or the amount of time it took to cause a denial of service. The Sitecom device will only accept a certain amount of subscriptions each time. The amount varies as different sleep values are being used. For example if the amount is 256, the Sitecom will stop responding after every 256 subscriptions being sent, causing a timeout around 45 seconds before continuing. Because of these timeouts it will take roughly 1,5 hour (despite the different sleep values) for the Sitecom router to cause a denial of service on UPnP, which is significantly more than the Edimax router.

When a denial of service is caused the device will still work, only UPnP will be completely down. This means that no event notification messages or responses to discovery messages are sent and no control, such as portmapping, will be possible. The only way to recover from this is to reset the router by powering down the device or disable and re-enable UPnP from the router's webinterface.

The denial of service might be caused for a few reasons. Both devices set a maximum number of subscriptions as resources allow. The maximum number will be set each time when UPnP is enabled. This number may vary when less resources are available. The maximum number of subscriptions is checked against the subscribers list when new subscriptions are accepted. When the subscribers list contain as much subscriptions as the maximum number of subscriptions allowed, no more subscriptions will be accepted (but also no subscriptions will be deleted).. However it is suspected that more resources are being used than initially were allowed. This can be due to the fact of the service accepting and handling too many subscriptions at once. When more resources are being used the maximum number of subscriptions that can be stored will be actually lower than the initial value, causing a premature failure of the service. This explains why the number of subscriptions is not consistent to cause a denial of service.

5 Conclusion

No harmful vulnerabilities were found in the UPnP eventing code in Intel's libupnp. By using UUIDs for communication between the publisher and control points a cancellation can only be performed by the subscriber itself and cannot easily cancel the subscription for another control point, without going through a lot of effort (sniffing the UUID). In case a different Callback URL is used by the subscriber only three entities should be aware of that unique UUID (control point, subscriber and publisher). However when sniffing is successful a subscription is easily cancelled, because the source of the unsubscription is not checked. The publisher only checks the UUID of the unsubscription message and matches it against its own entries of UUIDs. Testing showed that Callback URLs are not checked. This causes no real threat to websites, because they will reply with an error response. The fact that such subscriptions are not deleted from the subscribers list is in contradiction to Intels own specifications: "No unnecessary consuming of resources". A more intolerable point is the denial of service, which can be generated by sending as many subscriptions, until the service cannot handle any more.

6 Future research

Further research into this subject may focus on testing other UPnP stacks/devices. Vendors should have implemented the amount subscriptions it can maintain and deliver. Testing multiple devices of different vendors and UPnP stacks should show if this is the case.

References

[1] Universal Plug and Play: *website*, http://www.upnp.org/

[2] JINI *website*, http://www.jini.org/

[3] Zero Configuration Networking *website*, http://www.zeroconf.org/

[4] UPnP forum: *Website*, http://www.upnp.org/

[5] Sitecom router WL-534 *website*, http://www.sitecom.com/drivers_result.php?groupid=5&productid=522&version=CA;001

[6] Edimax BR-6104 router *website*, http://www.edimax.com/en/support_detail.php?pl1_id=3&pl1_idSelect=support.php%3Fpl1_id%3D3%26mwsp%3D1&pd_id=144

[7] Intel UPnP stack *website*, http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/overview/index.htm

[8] UPnP hacks by Armijn Hemel. *website/paper*, http://www.upnp-hacks.org/, http://www.upnp-hacks.org/sane2006-paper.pdf

[9] Hacking the interwebs - Reseach about UPnP by Adrian Pastor and Petko D. Petkov. *website*, http://www.gnucitizen.org/blog/hacking-the-interwebs/

[10] Presentation about the Crazy Toaster. *presentation*, http://sec.drorshalev.com/dev/upnp/DC-15-Shalev-005.ppt

[11] Presentation of Jonathan Squire on UPnP vulnerabilities. *presentation*, http://nchovy.kr/uploads/3/302/D1T2%20-%20Jonathan%20Squire%20-%20A%20Fox%20in%20the%20Hen%20House.pdf

[12] UPnP Design by Example, by Michael Jeronimo and Jack Weast *book*, ISBN-13: 978-0971786110

[13] Intel's UPnP Library documentation *website*, http://sourceforge.net/project/showfiles.php?group_id=7189&package_id=74114

[14] Edimax code archive *website*, http://edimax.nl/images/Image/products/BR-6104K/BR-6104K_GPL.tar.gz

[15] Sitecom code archive *website*, http://www.sitecom.com/documents/WL-160_GPL_11282006.tgz

[16] Miranda UPnP debugging tool *website*, http://www.sourcesec.com/2008/11/07/miranda-upnp-administration-tool/