A New Twist In SSDP Attacks

June 2018
A New Twist In SSDP Attacks

Executive Summary

Arbor ASERT has uncovered a new class of SSDP abuse where naïve devices will respond to SSDP reflection/amplification attacks with a non-standard port. The resulting flood of UDP packets have ephemeral source and destination ports, making mitigation more difficult - a SSDP diffraction attack. This behavior appears to stem from broad re-use in CPE devices of the open source library libupnp. Evidence from prior DDoS events suggest that attackers are aware of this behavior and may choose a pool of these misbehaving victims based on the efficacy of their attack. Using Arbor products to mitigate these attacks require inspecting packet content to filter the flood of SSDP replies and non-initial fragments.

Key Findings

- SSDP has been abused for reflection/amplification attacks for many years. In 2015, Arbor identified attacks utilizing SSDP traffic from ephemeral source ports.
- SSDP diffraction attacks that use ephemeral ports can defeat naïve port filtering mitigations.
- Surprisingly, the majority of the roughly 5 million SSDP servers reachable via the public Internet will respond from an ephemeral source port.
- The behavior stems from use of the open source library libupnp, which appears to be used in a variety of CPE devices.
- Defending against SSDP diffraction attacks requires inspecting packet content.

An Oversimplified Introduction To SSDP

SSDP (Simple Service Discovery Protocol) is a simple protocol designed to solve the problem of service discovery over a local network. The technology uses text-based HTTP messages over UDP (aka HTTPU) on the well-known port 1900. A client wishing to query for available services will issue a M-SEARCH command via HTTPU. Figure 1 shows a client querying the network via multicast for available services, and a printer replying with details of three services in three response packets.
On the wire, the M-SEARCH packet is almost always static, no matter the intent of the client. Refer to the UPnP specification for details about the meaning of these fields.

```
M-SEARCH * HTTP/1.1
HOST:239.255.255.250:1900
MAN: "ssdp:discover"
MX: 2
ST: ssdp:all
```

The SSDP server will respond with one or more HTTPU responses, one for each unique service that is available. UDP packets may contain multiple HTTPU responses separated by two carriage-return / newline characters.

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=120
ST: urn:schemas-upnp-org:device:WANDevice:1
USN: uuid:fc4ec57e-b051-11db-88f8-0060085db3f6:urn:schemas-upnp-org:device:WANDevice:1
EXT:
SERVER: Net-OS 5.xx UPnP/1.0
LOCATION: http://192.168.0.1:2048/etc/linuxigd/gatedesc.xml
```

An important field to understand for later discussion is the USN, or Unique Service Name. It is a UUID (Universally Unique Identifier) used to uniquely identify a device or service, although in practice UUIDs are re-used for entire device classes.

The LOCATION field is essential to the client - it points to a URL (TCP-based HTTP, not HTTPU) where the client can retrieve an XML-based description of the capabilities of the service. Some services will specify a SOAP (Simple Object Access Protocol) endpoint that clients can use to interact with the service, such as sending a job to a printer.

Other HTTPU verbs exist to manage discovery and graceful timeout of SSDP services but are not germane to this discussion.

**Reflection/Amplification**

The SSDP protocol is rife for Distributed Denial of Service (DDoS) abuse for one simple reason – many SSDP server implementations will answer requests sent to unicast addresses (e.g. 1.1.1.1), not just the well-known link-local multicast addresses for SSDP (239.255.255.250). Packets with multicast addresses as source or destination will not be routed via the Internet, but unicast addresses will. A roughly 100-byte request packet over UDP can yield as many as a dozen or more UDP responses (one for each service), all without the hurdle of having to setup a three-way TCP handshake. The resulting multiplicative effect of the responses will overwhelm the target. SSDP-based reflection/amplification attacks became fashionable in 2014, despite being well-understood long before then.

An attacker wishing to abuse SSDP for DDoS would take the following steps:

- Scan some portion of the Internet looking for IP addresses that respond to M-SEARCH queries with multiple (as many as possible!) packets.
- Using this list of abuse-able addresses, send a flood of M-SEARCH queries *with the source address spoofed to be the intended target*.
- Tweak the number of spoofed M-SEARCH packets until the target is overwhelmed.
Mitigating SSDP reflection/amplification attacks is straightforward since the attack packets originate from source port 1900, with an ephemeral destination port from the original spoofed request and contain an HTTPU response. Almost all uses of SSDP occur on the local network, and most large organizations don’t rely on the protocol for mission-critical applications, so packets with a UDP/1900 source port can generally be filtered at network boundaries during a crisis.

Abnormal SSDP Attacks

Both the attack and defense of SSDP reflection/amplification attacks have been well understood for years. But, a handful of DDoS attacks ASERT observed in 2015 exhibited different characteristics than a vanilla SSDP attack. The attack traffic did consist of HTTPU responses solicited from spoofed requests, but both the source and destination ports were ephemeral. The attack was a flood of UDP packets with high-numbered ports as the source and destination, rendering traditional source port filtering ineffective.

What was interesting about the attacks are that they started as a flood of UDP/1900 source port HTTPU packets (a normal attack), but when port filtering was put in place, the attack shifted to HTTPU packets with ephemeral sources (a diffraction attack). Clearly either the attacker, or the author of the attack tool, was aware of the difference in efficacy of both the normal attack and the diffraction attack.

What’s happening here? Let’s dig in.

Scanning

To understand how the SSDP population behaves, in September 2017 ASERT scanned the entire Internet. Every public address on the Internet was sent a generic M-SEARCH packet, and the first response packets recorded. The M-SEARCH packet used a static UDP source port of 1901 as a proxy for an ephemeral port, so both behaving and misbehaving (see below) responses could be recorded. The average “hit-rate” where a query was answered was 0.14%, or just over 5 million responses. This comprehensive scan yielded more responses than our later scan, which had a more aggressive timeout.

Analysis

For the purposes of this discussion, we’ll divide the population of responses into two categories. The behaving group is the set of addresses that, when sent an M-SEARCH query to UDP port 1900, responded with one or more UDP packets with a source port of 1900. Conversely, the misbehaving group is the set of addresses that when sent an M-SEARCH query to UDP port 1900 responded with one or more UDP packets with a source port other than 1900.

The first thing that immediately stands out, is that the population of misbehaving sources actually outnumbers those of behaving!

![Figure 2: Misbehaving / Behaving Population](image)

What can we learn about what makes the misbehaving group different than the behaving? The behaving group will all have source ports of 1900, but we can examine a histogram of the source ports from the misbehaving group.
A New Twist In SSDP Attacks

Different operating systems choose ephemeral source ports from different ranges. Windows usually chooses lower port numbers (1,025-5,000), and we observe a handful that fit that description. Linux usually chooses from a larger pool (32,768-61,000), and there are many more here than the Windows group.

The following table provides geo-location information for both the behaving and misbehaving populations:

<table>
<thead>
<tr>
<th>Country</th>
<th>Count</th>
<th>Country</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>628,121</td>
<td>China</td>
<td>585,104</td>
</tr>
<tr>
<td>Argentina</td>
<td>288,151</td>
<td>Russia</td>
<td>298,161</td>
</tr>
<tr>
<td>Russia</td>
<td>202,404</td>
<td>Vietnam</td>
<td>217,969</td>
</tr>
<tr>
<td>South Korea</td>
<td>139,623</td>
<td>South Korea</td>
<td>216,061</td>
</tr>
<tr>
<td>Taiwan</td>
<td>96,162</td>
<td>Venezuela</td>
<td>213,711</td>
</tr>
<tr>
<td>USA</td>
<td>89,452</td>
<td>Turkey</td>
<td>117,891</td>
</tr>
<tr>
<td>Italy</td>
<td>60,408</td>
<td>Algeria</td>
<td>108,088</td>
</tr>
<tr>
<td>India</td>
<td>56,592</td>
<td>Ukraine</td>
<td>101,320</td>
</tr>
<tr>
<td>Brazil</td>
<td>54,603</td>
<td>Japan</td>
<td>92,460</td>
</tr>
<tr>
<td>Tunisia</td>
<td>51,178</td>
<td>Greece</td>
<td>89,071</td>
</tr>
</tbody>
</table>

Figure 3: Histogram of Misbehaving Source Ports

Figure 4: Top Ten Behaving/Misbehaving Countries

USN is a UUID that supposedly uniquely defines each specific device, although in practice entire device classes re-use this field. Unfortunately, there is no canonical registry mapping USNs to devices, but a naïve count does offer some clues.
A New Twist In SSDP Attacks

The majority of behaving USNs are either 8c80f73f-4ba0-45fa-835d-042505d052be (an Internet Gateway Device - mostly likely a CPE router), fc4ec57e-b051-11db-88f8-0060085db3f6 (used by the MiniUPNP open source project), not specified, or zeroed out.

Of the misbehaving population, the 75802409-bccb-40e7-8e6c-fa095ecce13e USN and its variants are by far the most popular. More on that in the next section.

Other optional fields can be used in an HTTPU response, including a Server header. Much like the HTTP counterpart, the Server header identifies the type of server responding to the request.
A New Twist In SSDP Attacks

The behaving Server responses are all over the map, but there is a clear pattern to the misbehaving side—Linux/[kernel version], UPnP/1.0 Portable SDK For UPnP devices/[library version]. Several different kernel and library versions are represented. The library is a major clue as to the identity of the misbehaving population.

The final HTTPU response field we’ll examine is the X-User-Agent. Oddly enough, some responses will contain a X-User-Agent similar to a normal HTTP header’s User-Agent field.

<table>
<thead>
<tr>
<th>Behaving</th>
<th>Misbehaving</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X-User-Agent</strong></td>
<td><strong>Count</strong></td>
</tr>
<tr>
<td>&lt;none in initial response packet&gt;</td>
<td>2,158,308</td>
</tr>
<tr>
<td>redsonic</td>
<td>8,009</td>
</tr>
<tr>
<td>UPnP/1.0 DLNADOC/1.50</td>
<td>2</td>
</tr>
<tr>
<td>VisiMAX {8.03.00.00}</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The obvious pattern here is that the misbehaving set overwhelmingly contains redsonic. While the behaving set has a small handful of the same, the vast majority don’t include it.

Linux UPNP

The UUID that appears in misbehaving sources, but is almost completely absent from normal sources is 75802409-bccb-40e7-8e6c-fa095ecce13e. When searching for information about these long opaque UUIDs, you’ll normally only find information about DDoS attacks. But this UUID clearly belongs to the Linux UPNP Internet Gateway Device. From the source code linuxigd-1.0/etc/gatedesc.xml:

```
<device>
  <deviceType>urn:schemas-upnp-org:device:InternetGatewayDevice:1</deviceType>
  <friendlyName>Linux Internet Gateway Device</friendlyName>
  <manufacturer>Linux UPnP IGD Project</manufacturer>
  <manufacturerURL>http://linux-igd.sourceforge.net</manufacturerURL>
  <modelName>IGD Version 1.0.0</modelName>
  <UDN>uuid:75802409-bccb-40e7-8e6c-fa095ecce13e</UDN>
```
A New Twist In SSDP Attacks

The Linux UPNP Internet Gateway Device is an implementation of Microsoft's Internet Connection Service (ICS), a SSDP-aware protocol that solves the double-NAT problem. When two Internet users behind NAT devices wish to directly connect, for example to share a file without going through a third party, they can negotiate temporarily opening ports on both sides with the ICS protocol.

The Linux UPNP Internet Gateway Device only implements the ICS protocol, it relies on the Portable SDK for UPnP Devices (libupnp) for handling the lower-level UPNP. We installed the entire suite in our test lab to try and elicit the abnormal behavior. Here is a packet capture from our first normal M-SEARCH command:

<table>
<thead>
<tr>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>172.17.10.103</td>
<td>172.20.20.110</td>
<td>SSDP</td>
<td>135</td>
<td>M-SEARCH * HTTP/1.1</td>
</tr>
<tr>
<td>0.000338</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>345</td>
<td>53426 → 56847 Len=303</td>
</tr>
<tr>
<td>0.100466</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>345</td>
<td>53426 → 56847 Len=303</td>
</tr>
<tr>
<td>0.200509</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>354</td>
<td>53426 → 56847 Len=312</td>
</tr>
<tr>
<td>0.300745</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>354</td>
<td>53426 → 56847 Len=312</td>
</tr>
<tr>
<td>0.400917</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>417</td>
<td>53426 → 56847 Len=375</td>
</tr>
<tr>
<td>0.502486</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>417</td>
<td>53426 → 56847 Len=375</td>
</tr>
<tr>
<td>0.602720</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>389</td>
<td>44236 → 56847 Len=347</td>
</tr>
<tr>
<td>0.702888</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>389</td>
<td>44236 → 56847 Len=347</td>
</tr>
<tr>
<td>0.803132</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>354</td>
<td>48759 → 56847 Len=312</td>
</tr>
<tr>
<td>0.903307</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>354</td>
<td>48759 → 56847 Len=312</td>
</tr>
<tr>
<td>1.003470</td>
<td>172.20.20.110</td>
<td>172.17.10.103</td>
<td>UDP</td>
<td>393</td>
<td>48759 → 56847 Len=351</td>
</tr>
</tbody>
</table>

It exhibits the ephemeral port behavior by default! This dump was captured on the device running the Linux UPNP Internet Gateway, so no external network tampering like NAT is in effect.

Digging Into The Source

The libupnp project that actually implements the SSDP protocol is clearly of interest, so let's go even deeper and examine the source code, beginning with the latest version as of this writing 1.6.22 (released May 2017). Beginning with libupnp-1.6.22/upnp/src/ssdp/ssdp_server.c. The function readFromSSDPSocket() handles requests sent to UDP/1900:

```c
737     byteReceived = recvfrom(socket, requestBuf, BUFSIZE - (size_t)1, 0, 
738           (struct sockaddr *)&ss, &socklen);
```
It does some basic validation of the M-SEARCH request, eventually spawning another thread to handle the response ending up in the function `NewRequestHandler()` in `libupnp-1.6.22/upnp/src/ssdp/ssdp_device.c`:

```c
static int NewRequestHandler(  
    /* [in] Ip address, to send the reply. */  
    struct sockaddr *DestAddr,  
    /* [in] Number of packet to be sent. */  
    int NumPacket,  
    /* [in] . */  
    char **RqPacket)
{
```

The parameter `DestAddr` contains the presumably spoofed source address and the ephemeral port which sent the request. This crucial issue here is that the response creates a new socket, resulting in a new ephemeral source port that is not 1900:

```c
ReplySock = socket((int)DestAddr->sa_family, SOCK_DGRAM, 0);
```

The code continues to incorrectly assume it will respond to a unicast M-SEARCH request by setting the socket to multicast and TTL (time-to-live) to 4, without any error checking.

```c
int ttl = 4;
```

```c
setsockopt(ReplySock, IPPROTO_IP, IP_MULTICAST_IF,  
        (char *)&replyAddr, sizeof(replyAddr));
setsockopt(ReplySock, IPPROTO_IP, IP_MULTICAST_TTL,  
        (char *)&ttl, sizeof(int));
```

It finally packages up all the responses and sends them to the destination:

```c
rc = sendto(ReplySock, *(RqPacket + Index),  
        strlen(*(RqPacket + Index)), 0, DestAddr, socklen);
```

So why did the potentially mitigating TTL value of 4 fail to protect us? For that answer, we have to dive into the Linux kernel.

The call to `setsockopt()` did succeed, but the Linux kernel regards that only as a suggestion. In `linux/net/ipv4/ip_output.c` ([link](link)), the function `__ip_make_skb()` is responsible for delivering IP datagrams.
A New Twist In SSDP Attacks

First it will check cork, which ensures that IP fragments use the same TTL, and is not applicable here. Next the value inet->mc_ttl is taken from our earlier call to setsockopt(), but notice that it will only use this if the routing table entry for this particular destination is RTN_MULTICAST (aka, a multicast address). The response to the M-SEARCH request is being sent to a unicast address, so this check fails and falls through to use the default TTL!

The Server: Linux/[kernel version], UPnP/1.0 Portable SDK For UPnP devices/[library version] HTTPU header was found almost entirely in the misbehaving set. Sure enough, libupnp sets this by default, and even accounts for the Unspecified, UPnP/1.0, Unspecified header as the fourth largest in the misbehaving set seen in figure 7:

Finally, remember the X-User-Agent: redsonic HTTPU header enormously prevalent in the misbehaving set? It also comes from libupnp. In libupnp-1.6.22/upnp/src/inc/ssdplib.h:
libupnp includes this X-User-Agent by default, although it can be changed at compile-time by users of the library.

We believe devices that use libupnp are responsible for the aberrant DDoS behavior we have observed for the following reasons:

- The UUID 75802409-bcccb-40e7-8e6c-fa095ecce13e over represented in the misbehaving set is the Linux UPNP Internet Gateway Device, which uses libupnp.
- libupnp creates a new socket for responses, resulting in a new ephemeral port.
- The unique Server HTTPU header, hugely skewed towards the misbehaving set is the default value in libupnp.
- The X-User-Agent: redsonic HTTPU header, also vastly over represented in the misbehaving set is used by default in libupnp.

Mitigation

There is a strong argument for making SSDP servers only respond to multicast requests, but this is not how it works today. Making such a major change could subtly break misbehaving clients that depend on the behavior. Possibly the easiest fix, for behaving and misbehaving SSDP servers is to correctly set a small TTL on all reply packets. Reflection/amplification traffic would still make it out a few hops, but would not be effective enough to use in a real-world DDoS attack.

Conclusion

Attacks will always incrementally evolve just enough evade defenses. In this case we identified an effective new twist on an old, well-understood attack type. This revelation reminds us that defenders must constantly be aware of evolving attack methods and be as adaptable as the attackers. This specific attack highlights two trends we see time again: old code containing bugs being re-used in new consumer products, and subsequent exposure of those vulnerable populations.