Security Now! #846 - 11-23-21
HTTP Request Smuggling

This week on Security Now!

We’re going to start off this week by taking a careful look at a shocking proposal being made by the Internet's Engineering Task Force, the IETF. They’re proposing a change to a fundamental and long-standing aspect of the Internet’s routing which I think must be doomed to fail. So we’ll spend a bit of time on this in case it might actually happen. Then, Microsoft reveals some results from their network of honeypots, and we update on the progress, or lack of, toward more secure passwords. GoDaddy suffers another major intrusion, and just about every NetGear router really does now need to receive a critical update for the 5th time this year. This one is very worrisome. Then we're going to finish by winding up our propeller beanie-caps to explore the emerging threat represented by “HTTP Request Smuggling” — which everyone will understand by the time we’re done. It’s a sneaky way of slipping sensitive and damaging web requests past perimeter defenses. Oh! ... and on this November 23rd, a special Happy Birthday wish to Evan Katz, a long time friend and follower of this podcast.

Where there’s a will...

An idea whose time has passed...

As we know, there are several ranges of IPv4 addresses that are formally designated as unroutable. Three ranges of addresses were originally specified and set aside by RFC 1918 back in 1996. The smallest of these three is the one which most consumer routers use for their default local LAN, beginning with 192.168 often with dot 0 or dot 1. Consequently, any IPv4 address beginning with 192.168 is considered to be part of a private local network and no public routers will forward packets addressed to those IPs, anywhere. In the case of 192.168.x.x this creates a nice block of 64K IPs. The middle size RFC 1918-reserved private network is 172.16 through 172.31, which provides a network having 20 bits for specifying the machine on the network, or 1 million machines. And the largest one is the “10 dot” network which provides 24 bits for specifying a specific machine, or 16 million machines on the local network.

As we know, for quite some time there were so many IPv4 IPs that some were never allocated. Back when we first were using the Hamachi virtual network system, it was cleverly using the “5 dot” network since no one ever had. So it was able to confidently route any packets destined for “5 dot anything” through its own virtual network layer. But with time, the IPv4 space became depleted and “5 dot” was eventually opened for allocation. And there were also some original Internet-connected organizations that gave up their massive IPv4 allocations as the pressure for routable IPv4 addresses increased.

Another set-aside block of IPv4 space are all those addresses beginning with 127. Anyone who has spent much time poking around with IP networking on any IP-enabled operating system will have encountered the concept of the local loopback IP or loopback network. By universal convention and explicit specification, 127.0.0.1 always refers to the local machine. You can think of 127.0.0.1 as an alias for the local machine’s IP. If you “ping” 127.0.0.1 while any portion of your local network is running, that pinging will succeed, because it causes the machine to ping itself. No packets go anywhere. Many developers running a local web or other server on a machine will bind that server’s IP service to some port on 127.0.0.1 which makes its services available to any client running on that machine at the IP address 127.0.0.1.

Although only 127.0.0.1 is typically used, the entire “127 dot” network of 16 million IPs has been set aside as a local loopback network. If you open a Windows command prompt and enter the command: “route print” you’ll receive a dump of the system’s current routing table:

```
IPv4 Route Table
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0    10.10.255.254      10.10.0.10    266
       10.10.10.0      255.255.0.0         On-link       10.10.0.10    266
      10.10.10.10  255.255.255.255         On-link       10.10.0.10    266
   10.255.255.255  255.255.255.255         On-link       10.10.0.10    266
        127.0.0.0        255.0.0.0         On-link        127.0.0.1    306
        127.0.0.1  255.255.255.255         On-link        127.0.0.1    306
  127.255.255.255  255.255.255.255         On-link        127.0.0.1    306
                                     [...]
```

That routing table will contain a specific entry for 127.0.0.1 with a netmask of all 1's, meaning that packets exactly machine that IP are to be routed to the local interface. But you'll also find an entry for 127.0.0.0 with a netmask of 255.0.0.0, meaning that the last 24 bits are "match any" so that any packets whose destination IP begins with 127 will similarly be kept local. This is the way local machine IP stacks have always been configured as specified by RFC 1122 all the way back in 1989. On page 31 of that RFC it very clearly says...

```
(g)  { 127, <any> }
       Internal host loopback address.
       Addresses of this form MUST NOT appear outside a host.
```

And that explains why I did a serious double-take, and really did check the date to be certain it didn't say April 1st when, last week, I encountered an official IETF Standards Track proposal titled: **"Unicast Use of the Formerly Reserved 127/8"**
https://www.ietf.org/id/draft-schoen-intarea-unicast-127-00.html

Unicast is the formal name for standard packet — as opposed to broadcast or multicast. This pending IETF proposal is suggesting that the definition of the "127 dot" class A network should be changed to allow most of its "127 dot" space to be publicly routable in order to provide nearly 16 million more IPv4 addresses. The Abstract for this insanity says:

This document redefines the IPv4 local loopback network as consisting only of the 65,536 addresses 127.0.0.0 to 127.0.255.255 (127.0.0.0/16). It asks implementers to make addresses in the prior loopback range 127.1.0.0 to 127.255.255.255 fully usable for unicast use on the Internet.

Since I think this is so interesting, and since I'm sure that any of this podcast's listeners who are aware of Internet engineering are currently picking themselves up off the floor, I'm going to share a few more of the good bits from the proposal...

1. Introduction

With ever-increasing pressure to conserve IP address space on the Internet, it makes sense to consider where relatively minor changes can be made to fielded practice to improve numbering efficiency. One such change, proposed by this document, is to allow the unicast use of more than 16 million historically reserved addresses in the middle of the IPv4 address space.

This document provides history and rationale to reduce the size of the IPv4 local loopback network ("localnet") from /8 to /16, freeing up over 16 million IPv4 addresses for other possible uses.

When all of 127.0.0.0/8 was reserved for loopback addressing, IPv4 addresses were not yet recognized as scarce. Today, there is no justification for allocating 1/256 of all IPv4 addresses for this purpose, when only one of these addresses is commonly used and only a handful are regularly used at all. Unreserving the majority of these addresses provides a large number of additional IPv4 host addresses for possible use, alleviating some of the pressure of IPv4 address exhaustion.

2. Background

The IPv4 network 127/8 was first reserved by Jon Postel in 1981 [RFC0776]. Postel's policy was to reserve the first and last network of each class, and it does not appear that he had a specific plan for how to use 127/8. Apparently, the first operating systems to support a loopback interface as we understand it today were experimental Berkeley Unix releases by Bill Joy and Sam Leffler at the University of California at Berkeley. The choice of 127.0.0.1 as loopback address was made in 1983 by Joy and Leffler in the code base that was eventually released as 4.2BSD. Their earliest experimental code bases used 254.0.0.0 and 127.0.0.0 as loopback addresses. Three years later, Postel and Joyce Reynolds documented the loopback function in November 1986 [RFC0990], and it was codified as a requirement for all Internet hosts three years after that, in [RFC1122].

The substantive interpretation of these addresses has remained unchanged since RFC 990 indicated that the network number 127 is assigned the "loopback" function, that is, a datagram sent by a higher level protocol to a network 127 address should loop back inside the host. No datagram "sent" to a network 127 address should ever appear on any network anywhere. Many decisions about IPv4 addressing contemporaneous with this one underscore the lack of concern about address scarcity. It was common in the early 1980s to allocate an entire /8 to an individual university, company, government agency, or even a research project.

By contrast, IPv6, despite its vastly larger pool of available address space, allocates only a single local loopback address (::1) [RFC4291]. This appears to be an architectural vote of confidence in the idea that Internet protocols ultimately do not require millions of distinct loopback addresses.

Most applications use only the single loopback address 127.0.0.1 ("localhost") for IPv4 loopback purposes, although there are exceptions. For example, the systemd-resolved service on Linux provides a stub DNS resolver at 127.0.0.53.

In theory, having multiple local loopback addresses might be useful for increasing the number of distinct IPv4 sockets that can be used for inter-process communication within a host. The local loopback /16 network retained by this document will still permit billions of distinct concurrent loopback TCP connections within a single host, even if both the IP address and port number of one endpoint of each connection are fixed.

3. Change in Status of Addresses Within 127/8

The purpose of this document is to reduce the size of the special-case reservation of 127/8, so that only 127.0/16 is reserved as the local loopback network.

Other IPv4 addresses whose first octet is 127 (that is, the addresses 127.1.0.0 to 127.255.255.255) are no longer reserved and are now available for general Internet unicast use, treated identically to other IPv4 addresses, and subject to potential future allocation.

All host and router software SHOULD treat 127.1.0.0 to 127.255.255.255 as a global unicast address range.

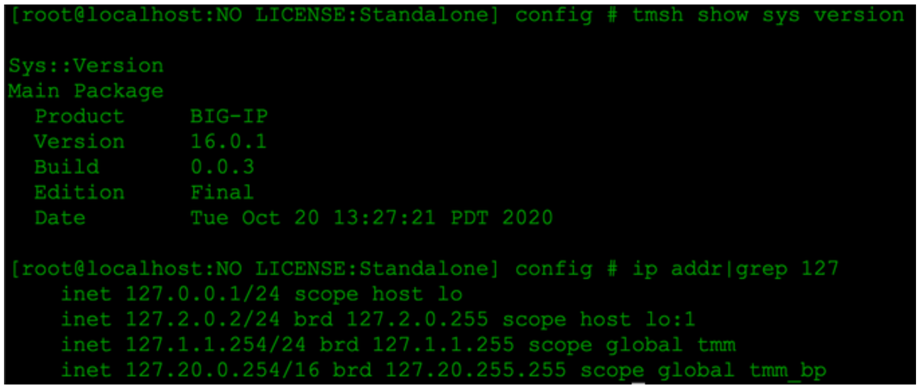
Clients for auto-configuration mechanisms such as DHCP [RFC2131] SHOULD accept a lease or assignment of addresses within 127.1/16 to 127.255/16 whenever the underlying operating system is capable of accepting it. Servers for these mechanisms SHOULD assign this address when so configured.

Now, in this case, I'm not even going to rhetorically ask "what could possibly go wrong" because the question doesn't need to be asked. Just think of all the many millions of existing embedded TCP/IP stacks in all the many millions of existing IoT devices, that sold and shipped with what has, since the dawn of the Internet, been a default local routing table that will never forward any packet starting with 127. As we know, none of those devices will ever be updated or changed. And what of all the many millions of SOHO routers running embedded Linuxes that will also never be updated and will, similarly, never forward any packets starting with 127.

The only way to read this is as an extreme desperation move and an appreciation of just how badly no one wants to move to IPv6. It's going to be interesting to see what happens to this proposal. Those of us who see the folly in this are not alone. Here's a sampling of nine tweets from just one thread on Twitter:

Ben Aveling / @BenAveling
Was it a mistake to allocate an entire A class to the Loopback range? Yes, yes it was.
Is it too late to fix that mistake? Yes, yes it is.

Rich Mirch / @0xm1rch
Real products use it. For example F5 BIG-IP has several subnets under 127/8. loopback is configured for /24



```
[root@localhost:NO LICENSE:Standalone] config # tmsh show sys version

Sys::Version
Main Package
  Product     BIG-IP
  Version     16.0.1
  Build       0.0.3
  Edition     Final
  Date        Tue Oct 20 13:27:21 PDT 2020

[root@localhost:NO LICENSE:Standalone] config # ip addr|grep 127
    inet 127.0.0.1/24 scope host lo
    inet 127.2.0.2/24 brd 127.2.0.255 scope host lo:1
    inet 127.1.1.254/24 brd 127.1.1.255 scope global tmm
    inet 127.20.0.254/16 brd 127.20.255.255 scope global tmm_bp
```

Andy90 / @Andy_ninety
TLS1.2 was around for so long that most middlebox vendors didn't handle handshakes appropriately, thus TLS1.3 has to masquerade as TLS1.2. I dread to think how many network tools simply are hardcoded and untested for anything like this and would break routing on the spot.

Jim Kladis / @JimKladis
I've seen 127 addresses in backbone hops.
There's much better waste to go after.
IBM: /8
HPE: /8
Xerox (Xerox???): /8
Then the tech universities and anything the DoD still has.

Luca Francesca / @Flukas88
If only we had an alternative like… dunno… IPv6? I know I know … bleeding edge stuff, its only 20y old

One Matt among many / @0xMatt
Thinking of the number of firewall rules I've seen, by default, every one of them, which "Allow 127/8". Please just use IPv6 already

Pascal Ernster / @srslypascal
*Before even *considering* embarking on this journey of nonsense, they should reassign a dozen of the 14 /8s that the DoD has. And when those have been reassigned, continue with the /8s that are currently assigned to Apple, Ford, Prudential and the USPS. Heck, I'd even go as far as saying that *all* /8s should be assigned to regional Internet registries.*

Jonathan Katz / @katzmandu
Other than security there is the practicality of this; I'm sure there is LOTS of software (legacy and otherwise) out there with 127/8 hard-coded in it which would break BADLY.

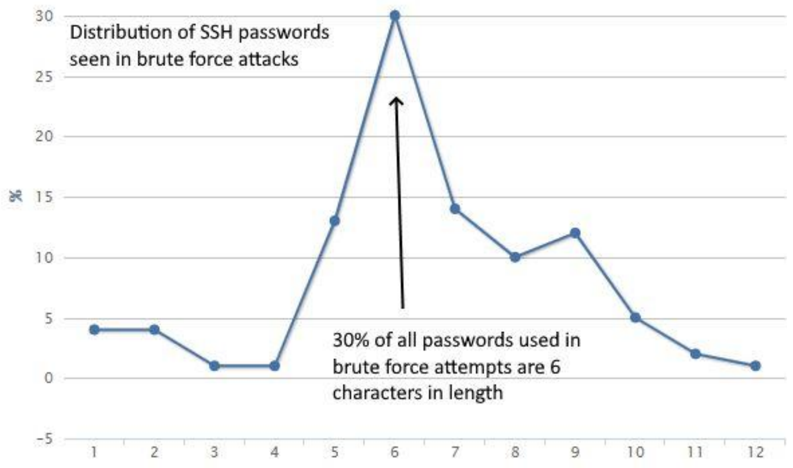
David / @hcetamd
Pretty much all operating systems boot up with a 127.0.0.0/8 loopback by default. Changing that would require updates to software/firmware (or changes to startup scripts). These newly-routable addresses would be inaccessible to many devices. Who would want them?

The points that were made about all of the other low-hanging /8 networks were interesting. It must be that companies see their legacy allocations as corporate assets. They must know that they'll never have any need for nearly the space they own. Many other much larger enterprises are getting along just fine with very modest allocations. And with IPv4 addresses currently trading at $36 each, a 16 million /8 allocation is currently valued at more than half a billion dollars. And since that price has been rising steadily, the value of those assets are likely to only increase over time. Having the non-profit US Department of Defense squatting on many /8 blocks is more difficult to defend.

But the counterpoint to all of this, as others in that thread noted, is IPv6... fully functional and ready to use.

Security News

The stats of brute force password attacks.



Microsoft actually has a job position known as "Head of Deception." The slot is currently occupied by a security researcher named Ross Bevington. Ross is the guy who puts pot of tasty honey out onto the Internet and collects all of the attempts made to get in.

Ross spent 30 days collecting more than 25 million brute force attacks against a tantalizing SSH server and he came away with a few interesting insights. He found that 77% — just over three quarters — of all attempts guessed a password between 1 and 7 characters. A password over 10 characters was only seen in 6% of cases. Only 7% of the brute-force attempts analyzed in the sample data included a special character. 39% had at least one number, and NONE of the brute-force attempts used passwords that included white space. Ross' conclusion is that attackers don't bother attempting to brute-forcing long passwords.

Ross said that based upon data from more than 14 billion brute-force attacks attempted against Microsoft's network of honeypot servers, through September of this year, attacks on Remote Desktop Protocol (RDP) servers have tripled compared to 2020, seeing a rise of 325%. Network printing services also saw an increase of 178%, as well as Docker and Kubernetes systems, which saw an increase of 110%.

The Most Common Passwords

And while we're on the topic of passwords, Nordpass just published their annual analysis of password use across 50 countries. The Top 10 most common passwords in 2021 currently are:

1. 123456 (103,170,552 hits) more than TWICE the #2 password
2. 123456789 (46,027,530 hits)
3. 12345 (32,955,431 hits)
4. qwerty (22,317,280 hits)
5. password (20,958,297 hits)
6. 12345678 (14,745,771 hits)
7. 111111 (13,354,149 hits)
8. 123123 (10,244,398 hits)

9. 1234567890 (9,646,621 hits)
10. 1234567 (9,396,813 hits)

Among their other findings, the researchers found that (in their words) "a stunning" number of people use their own name as their password ("charlie" appeared as the 9th most popular password in the UK). “Onedirection” was a popular music-related password option, and the number of times “Liverpool” appears could indicate how popular the football team is. But in Canada “hockey” was unsurprisingly the top sports-related option in active use. Swear words are also commonly employed, and when it comes to animal themes, “dolphin” was the most popular choice internationally. (So, Leo... it looks like our favorite “monkey” has fallen into disfavor.)

I’m somewhat surprised by those, since most are purely numeric, without any letters or special characters. Many websites would allow them to be used. But if they were in place before specific requirements began to be added, I suppose I can see how, as legacy passwords, they could still be valid today.

GoDaddy Breached Bigtime!

Yesterday, the well-know Internet domain registrar and more recently cloud hosting provider, GoDaddy said that a hacker gained access to the personal information of more than 1.2 million customers of its WordPress hosting service using a compromised password which gave the attacker access to the provisioning system in their legacy codebase for their Managed WordPress.

Being one of the world's largest domain registrars and a web hosting company providing services to more than 20 million customers worldwide, as well as being publicly traded, they needed to promptly inform the US Securities and Exchange Commission. In the SEC documents filed yesterday, GoDaddy stated that it discovered the breach last Wednesday after noticing “suspicious activity” on their Managed WordPress hosting environment.

Subsequent investigation revealed that a hacker had unfettered access to GoDaddy's servers for more than two months, since at least September 6. And based upon the available evidence, the hacker had gained access to:

- Up to 1.2 million active and inactive Managed WordPress customers had their email addresses and customer numbers exposed.
- The original WordPress Admin password that GoDaddy issued to customers when a site was created.
- For active customers, sFTP and database usernames and passwords were exposed.
- For a subset of active customers, the SSL private key was exposed.

GoDaddy had already reset sFTP and database passwords exposed in the hack as well as the admin account password for customers who were still using the default one that GoDaddy had originally issued when their sites were created. GoDaddy is currently in the process of issuing and replacing new SSL certificates for affected customers. And they've notified law enforcement and are working with an IT forensics firm to further investigate the incident. Customer's were also notified of this yesterday.

And for those who are counting, this is not GoDaddy's first trouble with breaches...

Last May, GoDaddy alerted some of its customers that an unauthorized party used their web hosting account credentials in October of 2020 to connect to their hosting account via SSH. In that instance GoDaddy's security team discovered the breach after spotting an altered SSH file in GoDaddy's hosting environment and suspicious activity on a subset of GoDaddy's servers.

And back in 2019, scammers used hundreds of compromised GoDaddy accounts to create 15,000 subdomains, attempting to impersonate popular websites and redirect potential victims to spam pages offering bogus products.

And earlier in 2019, GoDaddy was found to be injecting JavaScript into US customers' sites without their knowledge, thus potentially rendering them inoperable or impacting their overall performance.

I registered grc.com in December of 1991... a few months after the microsoft.com domain was registered. And I chose Network Solutions since they were the original Internet registrar. But our long time listeners will know that I could finally no longer tolerate their slimy upselling tactics. It was necessary to say “no” to various offers over and over again, and more annoyingly, to carefully uncheck various enabled-by-default additional cost services which other registrars were by then offering for free. So I started to feel as though my loyalty had become misplaced and I decided that I had to move. I chose Hover (also a sponsor of the TWiT Network) and I’ve never looked back. When I was making the switch, I considered GoDaddy. Mark Thompson uses them and recommended them. But they were too brightly colored and hyper-commercial for my taste. Their website looked like a cartoon. That’s what I wanted to get away from. The last thing you want in a domain registrar is excitement. Things are rather exciting over at GoDaddy right now. No thank you. What you want from a domain registrar is a great deal of boredom.

A heads-up about NetGear routers

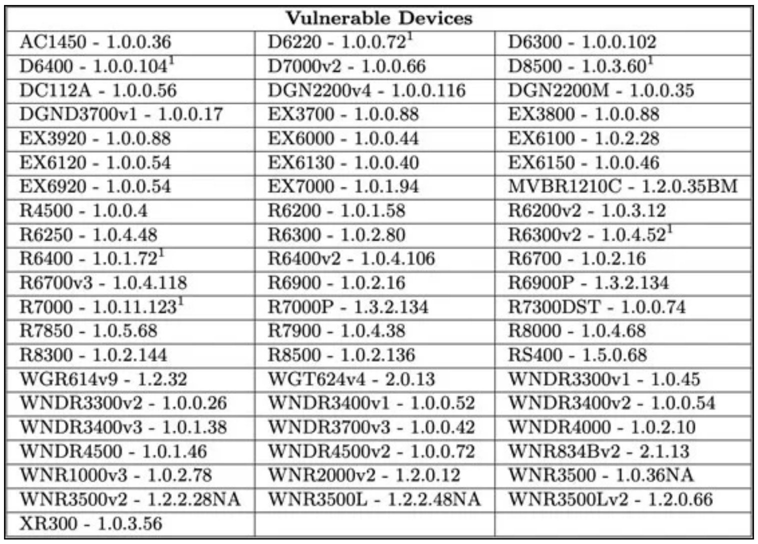
Last week, Netgear released a round of patches to remediate a high severity remote code execution vulnerability affecting 61 different models (I’ve included a table of the impacted routers in the show notes... but, really, all of our listeners who are using NetGear routers should make a point of checking in for any available update.)

It's another UPnP vulnerability which, when exploited, would allow remote attackers to take control of a vulnerable system. The good news is, most routers won't be exposing their UPnP port to the public Internet. On the other hand, GRC's UPnP Internet Exposure Test has counted 55,166 positive Internet-facing tests since I put it online. I'm not logging unique IPs, so there are doubtless multiple visits from the same user. But all of these routers will have UPnP running on their LAN interface where the exploitation could be by way of malicious script running on a browser inside the LAN — thus accessible to the router’s internal LAN interface — to then make the router remotely accessible. So, this should definitely not be taken lightly.

And in recognition of this severity, this vulnerability was assigned CVE-2021-34991 with a CVSS severity score of 8.8. It's a pre-authentication buffer overflow flaw affecting what appears to be pretty much all of NetGear’s small office and home office routers, and it can lead to remote code execution with the highest privileges.

The vulnerability stems from the fact that the UPnP daemon accepts, by design, unauthenticated HTTP SUBSCRIBE and UNSUBSCRIBE requests, which are event notification alerts that devices use to receive notifications from other devices when certain configuration changes, such as media sharing, occur on the network.

But there's a memory stack overflow bug in the code that handles the UNSUBSCRIBE requests (whoopsie!), which enables an adversary to send a specially crafted HTTP request and run malicious code on the affected device, including resetting the administrator password and delivering arbitrary payloads. As we know, HTTP requests are things that web browsers routinely generate. So the idea of JavaScript doing this is not far fetched. Once the password has been reset, the attacker can then login to the webserver and modify any settings or launch further attacks on the router's internal web server.



Vulnerable Devices		
AC1450 - 1.0.0.36	D6220 - 1.0.0.72[1]	D6300 - 1.0.0.102
D6400 - 1.0.0.104[1]	D7000v2 - 1.0.0.66	D8500 - 1.0.3.60[1]
DC112A - 1.0.0.56	DGN2200v4 - 1.0.0.116	DGN2200M - 1.0.0.35
DGND3700v1 - 1.0.0.17	EX3700 - 1.0.0.88	EX3800 - 1.0.0.88
EX3920 - 1.0.0.88	EX6000 - 1.0.0.44	EX6100 - 1.0.2.28
EX6120 - 1.0.0.54	EX6130 - 1.0.0.40	EX6150 - 1.0.0.46
EX6920 - 1.0.0.54	EX7000 - 1.0.1.94	MVBR1210C - 1.2.0.35BM
R4500 - 1.0.0.4	R6200 - 1.0.1.58	R6200v2 - 1.0.3.12
R6250 - 1.0.4.48	R6300 - 1.0.2.80	R6300v2 - 1.0.4.52[1]
R6400 - 1.0.1.72[1]	R6400v2 - 1.0.4.106	R6700 - 1.0.2.16
R6700v3 - 1.0.4.118	R6900 - 1.0.2.16	R6900P - 1.3.2.134
R7000 - 1.0.11.123[1]	R7000P - 1.3.2.134	R7300DST - 1.0.0.74
R7850 - 1.0.5.68	R7900 - 1.0.4.38	R8000 - 1.0.4.68
R8300 - 1.0.2.144	R8500 - 1.0.2.136	RS400 - 1.5.0.68
WGR614v9 - 1.2.32	WGT624v4 - 2.0.13	WNDR3300v1 - 1.0.45
WNDR3300v2 - 1.0.0.26	WNDR3400v1 - 1.0.0.52	WNDR3400v2 - 1.0.0.54
WNDR3400v3 - 1.0.1.38	WNDR3700v3 - 1.0.0.42	WNDR4000 - 1.0.2.10
WNDR4500 - 1.0.1.46	WNDR4500v2 - 1.0.0.72	WNR834Bv2 - 2.1.13
WNR1000v3 - 1.0.2.78	WNR2000v2 - 1.2.0.12	WNR3500 - 1.0.36NA
WNR3500v2 - 1.2.2.28NA	WNR3500L - 1.2.2.48NA	WNR3500Lv2 - 1.2.0.66
XR300 - 1.0.3.56		

The guy who discovered and reported the trouble noted that "Since the UPnP daemon runs as root, the highest privileged user in Linux environments, the code executed on behalf of the attacker will be run as root as well. With root access on a device, an attacker can read and modify all traffic that is passed through the device."

So, again, it's time for all owners of NetGear routers to make sure they're running the most recent firmware for their devices. :)

HTTP Request Smuggling

“HTTP request smuggling”, also sometimes called “HTTP Desync Attacks”, take advantage of the fact that much of today's modern Internet is far more complex than a web client connecting to a web server. Any website hosted by Cloudflare provides an example of just how complex the Internet has become.

Many of today's websites and web-hosted applications employ chains of HTTP servers between their users and the backend application logic. Users requests are first received by a front-end server. It might be performing web filtering, load balancing, reverse proxying and/or caching. Whatever the case, this front-end server forwards requests to one or more back-end servers. And not only is this type of architecture is increasingly common, in many cases it's fundamental to modern cloud-based applications.

For the sake of expediency and efficiency the front-end connections to back-end servers are persistent and long lived. This saves a huge amount of connection setup and teardown time. And the protocol is very simple, with the front-end sending HTTP requests one after another to the receiving server, which parses the HTTP request headers to determine where one request ends and the next one begins. In other words, to determine the request boundaries. And when this is being done it's crucial that the front-end and back-end systems agree about the boundaries between requests. Otherwise, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems... which is, of course, exactly what can be made to happen.

HTTP request smuggling is a slippery and tricky technique which deliberately manipulates the Content-Length and Transfer-Encoding headers of multiple HTTP requests in such as way that the various servers involved in servicing the requests become confused about the successive HTTP request boundaries in a way that allows clever attackers to achieve web cache poisoning, session hijacking, cross-site scripting and even web application firewall bypass.

And unlike many of the interesting technological attacks and tricks we often discuss here, these are not theoretical. Last week one of the vulnerabilities that was being exploited used HTTP Request Smuggling in part of its attack chain. These are the real deal.

I mentioned the Content-Length and Transfer-Encoding headers. HTTP request smuggling vulnerabilities arise because, unfortunately, the HTTP specification provides two different ways to specify where a request ends.

The Content-Length header is straightforward and by far the most common method. It simply specifies the length of the message body in bytes. For example:

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

q=smuggling
```

But the use of a Transfer-Encoding header is also completely valid and it, too, can be used to specify that the HTTP message body uses chunked encoding. When the Transfer-Encoding header specifies “chunked” this means that the message's body consists of one or more “chunks” of data where each chunk consists of the chunk size specified in hexadecimal bytes, followed by a newline, then the chunk contents and another newline. The message is terminated when the reader encounters a chunk of zero size. For example:

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

b
q=smuggling
0
```

Students of this podcast will immediately see how fragile this chunked encoding is. For one thing, it's doing a bit of interpretation. But the big No-No from an architectural design standpoint is that it is mixing control metadata with the data. We've previously seen how the immensely popular and powerful “printf” function, which is present in many coding languages, suffers from a similar design vulnerability by mixing data and control metadata within the same string. If an attacker can get control of the string, a great deal of mischief can be had.

As for HTTP, many researchers are unaware that chunked encoding is valid in HTTP requests because web browsers don’t normally use chunked encoding in their requests, and it is normally seen only in server responses. But it’s in the spec and many servers support it.

Since HTTP provides two entirely different methods for specifying the length of HTTP messages -- and one is quite fragile and susceptible to abuse, it's possible for a single request to use both methods and this can be done in such a fashion that they conflict with each other. This possibility was understood by HTTP's designers who attempted to prevent this problem by simply stating that if both the Content-Length and Transfer-Encoding headers are present, the Content-Length header should be ***ignored***. Whoops! The more fragile of the two wins in a contest. And it turns out that while this simple exclusion might be sufficient to avoid ambiguity when only a single server is in play, when two or more servers are chained together bad stuff can happen.

Bad stuff happens because some servers don't support the Transfer-Encoding header in requests at all and some servers that do support the Transfer-Encoding header can be induced not to process it if the header is obfuscated in some way.

And here's the key: If the front-end and back-end servers behave differently in relation to the (possibly obfuscated) Transfer-Encoding header, then they might disagree about the boundaries between successive requests... which enables request smuggling vulnerabilities.

So, request smuggling attacks are created by placing both the Content-Length header and the Transfer-Encoding header into a single HTTP request and manipulating them so that the front-end and back-end servers will process the request differently. And, naturally, the design of a successful attack depends on the behavior of the two servers. Attacks come in three forms depending upon the characteristics of the specific servers:

CL.TE: *When both headers are present, the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header.*

TE.CL: *When both headers are present, the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header.*

TE.TE: *When both headers are present, the front-end and back-end servers both support the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way.*

Let's take the first case of a CL.TE attack where, when both headers are present, the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

The front-end server processes the Content-Length header so it understands that the entire request body is 13 bytes long, right through the end of SMUGGLED. So this request is forwarded to the back-end server containing “SMUGGLED” in the content body of the request.

But the back-end server is more compliant with the HTTP specification. So it ignores the Content-Length header and instead processes the Transfer-Encoding header as it’s supposed to. It, therefore, treats the message body as using chunked encoding.

It processes the message's first chunk, which is just a 0 on a line by itself. So it treats that as the end of the first HTTPS request. And what follows, in our example the word “SMUGGLED” is seen as another entirely separate and distinct HTTP query.

Therefore, if our attacker had embedded some sort of useful query into the end of the first query, the front-end server would **not** have seen it and would have treated it as query data for the first query. But the back-end server **would** have seen that hidden embedded query as it own free-standing second query, and would have acted upon it.

Obviously, if the front-end server is examining requests and functioning as an HTTPS firewall, this technique slips a query to the backend right through the front-end firewall!

The reverse of this attack, the TE.CL can be used when the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header in the presence of both headers:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 3
Transfer-Encoding: chunked

8
SMUGGLED
0
```

Since the front-end server honors chunked encoding, it will interpret the message's two chunks as a single message and will forward the entire query to the back-end server. But the back-end server favors the use of content-length encoding. So when the attacker's query arrives it sees its length as 3 bytes consisting of the '8' and the line-terminating carriage return and line feed. But there's still more message, which the back-end server interprets as another incoming HTTP query.

So, just as before, an attacker is able to smuggle a query past the front-end server to have it honored by the back-end.

The final TE.TE vulnerability can often be effective when both servers understand chunked transfer encoding but parse query headers in subtly different ways. For example, if a space appears before the colon (:) after the header name, some servers will accept that sloppy formatting whereas others won't. Or if a tab character is used instead of the space to separate the header name from its value, again, some servers will accept the tab whereas others won't. And there are many such variations that actually work in the real world.

This is what I meant when I said that the chunked method of encoding was so fragile. All that's necessary is to find some way of expressing this Transfer-Encoding: chunked header in a way that the various servers in a chain will process it differently, obeying it or ignoring it, and that difference can be exploited to smuggle queries past the front-end system.

I considered taking this to the next step by demonstrating how this request smuggling can be leveraged into various forms of devastating web attacks. But a predominantly audio podcast makes that impractical, and I'm certain that anyone who has managed to follow this much will now have a good sense for the fundamentals of HTTP Request Sumuggling attacks.

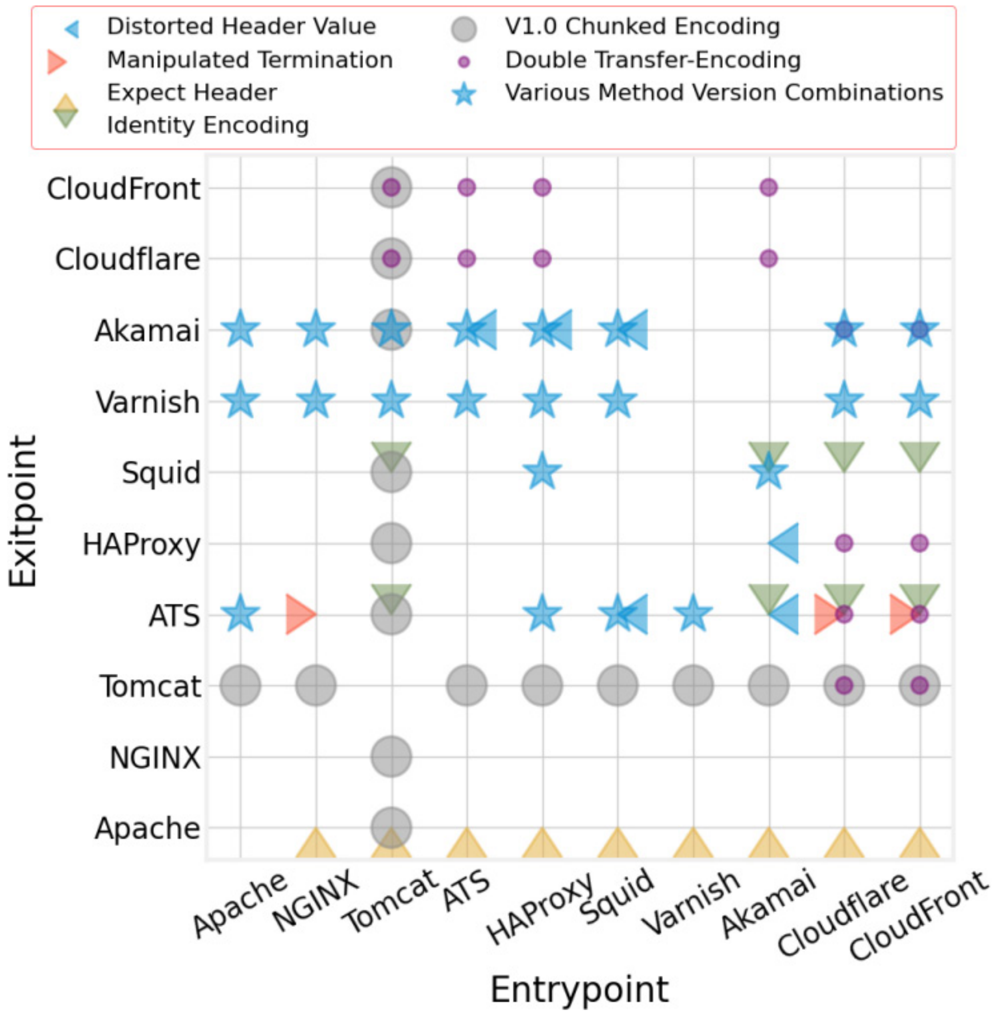
These potential problems were first discovered and documented back in 2005. But they seemed to be mostly theoretical and were viewed as being particularly valuable or practical. That's changed.

Researchers from Northeastern University and Akamai Technologies have written a paper titled: "T-Reqs: HTTP Request Smuggling with Differential Fuzzing" which was just presented during the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21).

Their paper's Abstract explains:

HTTP Request Smuggling (HRS) is an attack that exploits the HTTP processing discrepancies between two servers deployed in a proxy origin configuration, allowing attackers to smuggle hidden requests through the proxy. While this idea is not new, HRS is soaring in popularity due to recently revealed novel exploitation techniques and real-life abuse scenarios.

In this work, we step back from the highly-specific exploits hogging the spotlight, and present the first work that systematically explores HRS within a scientific framework. We design an experiment infrastructure powered by a novel grammar-based differential fuzzer, test 10 popular server/proxy/CDN technologies in combinations, identify pairs that result in processing discrepancies, and discover exploits that lead to HRS. Our experiment reveals previously unknown ways to manipulate HTTP requests for exploitation, and for the first time documents the server pairs prone to HRS.



This diagram shows what the researchers found when 10 popular servers, CDNs and proxies were "fuzzed" with a wide range of HTTP header mutations. One of the items is interesting: "Double Transfer-Encoding" suggests that duplicating headers can also create new vulnerabilities.

https://bahruz.me/papers/ccs2021treqs.pdf