WEB APPLICATION FIREWALL BYPASS VIA BLUECOAT DEVICE

Red Timmy Security

February 15th, 2020

Red Timmy Security

https://www.redtimmy.com/blog
https://www.twitter.com/redtimmysec

Summary

FOREWARD

This whitepaper is all about how to hack a company by circumventing its WAF (Web Application Firewall) through the abuse of a different security appliance and win bug bounties ☺

INTRODUCTION

Hey, wait! What do bug bounties and network security appliances have in common? Usually nothing! On the contrary, the security appliances allow virtual patching practices and actively participate to reduce the number of bug bounties paid to researchers…but this is a reverse story: a bug bounty was paid to us thanks to a misconfigured security appliance. We are not going to reveal neither the name of the affected company (except it was a Fortune 500) nor the one of the vulnerable component. However, we will be talking about the technique used, because it is astonishingly of disarming simplicity.

THE OBSERVATION

All has begun by browsing what at that time we did not even know yet to be a valuable target, let us call it "https://targetdomain". Almost by accident, we noticed that a subdomain responsible for the authentication on that website had exposed some CSS and Javascript resources attributable to a Java component well known to be vulnerable to RCE (Remote Code Execution).

The weird thing was that by browsing the affected endpoint (something like "https://**auth**.targetdomain/**vulnerable_endpoint**?param=malicious_RCE_payload") we received a HTTP 404 reply from the server, which made us suspect the presence of a Web Application Firewall. Why that particular endpoint should not be reachable if the resources decorating it (like *.css* and *.js* files) are? This clearly made us believe we were in front of a WAF. After a few more requests, all blocked, we confirmed some kind of WAF rule was indeed preventing us from reaching the target endpoint.

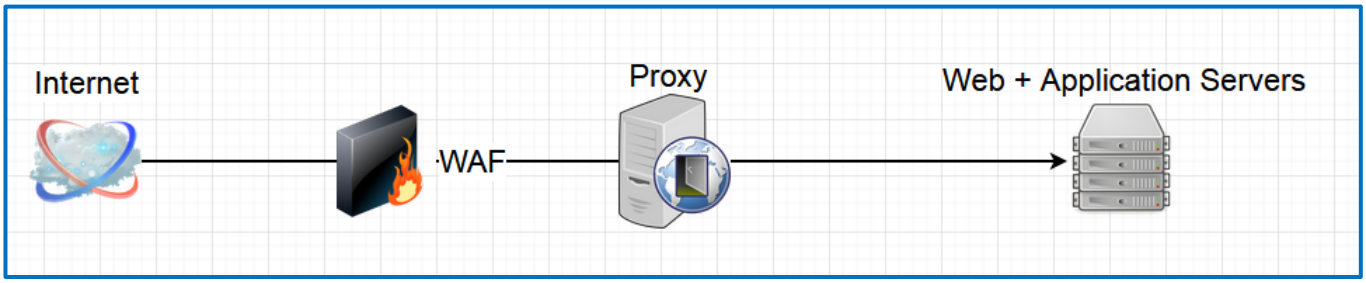
THE "WEIRD" THING

By browsing one of the applications hosted (i.e. https://targetdomain/**appname**) we are invited to authenticate to "https//**auth**.targetdomain". During the authentication process, we notice another strange thing. At a certain moment we are redirected to a URL like:

"https//targetdomain/?cfru=aHR0cHM6Ly90YXJnZXRkb21haW4vYXBwbmFtZQ==”

with "aHR0cHM6Ly90YXJnZXRkb21haW4vYXBwbmFtZQ==" being clearly a base64-encoded string. The base64 payload, after decoding, showed to be nothing more that the URL we had originally requested access to before starting the authentication, that is "https://targetdomain/appname".

But what actually that "cfru" parameter was? Some quick research online shows it is part of the Bluecoat web filtering technology, a notorious proxy server appliance. So, this told us a bit more about the remote infrastructure. The HTTP requests we send to the target cross at least one WAF device and a Bluecoat proxy server before reaching the front end web servers and application servers, like reconstructed below.



THE IDEA

A light bulb has lit up on our head once we discovered that this "cfru" parameter was publicly accessible, namely no authentication to the portal was required to pass our payload to it. Therefore we started to base64-encode URLs of external domains under our control and feed the "cfru" parameter with these strings. The hope was to trigger some kind of SSRF. What we got at the end was much better.

Unfortunately, at that specific moment in time, we did not receive back any HTTP requests. However, in our internet-exposed machines, we could see the DNS resolution process started from "targetdomain". It seemed the TCP outgoing connections from target website were prohibited. The only authorized thing was, as said, DNS traffic. Then instead of trying to trigger SSRF requests to external hosts we turned our attention to internal subdomains (https://auth.targetdomain, https://blog.targetdomain, https://www.targetdomain, etc...).

We start to base64-encode few of these URLs into the "`cfru`" parameter and almost immediately notice another weirdness. For some URLs we get a HTTP 302 redirect back. For some others we do not. In this latter case instead the HTTP body in the reply contains the HTML code of the requested resource, as if Bluecoat forwarded the request to the destination resource and returned its content back to us by acting as a web proxy. Most importantly, this behavior was observed also when we encoded in the "`cfru`" parameter the subdomain responsible for the authentication to the portal (https//**auth**.targetdomain), namely the one we believed was hosting a Java component vulnerable to RCE.

THE TURNING POINT

Here was the turning point! We have made the following assumption. If the resource

https://auth.targetdomain/vulnerable_endpoint?param=malicious_RCE_payload

is directly browsed, our HTTP request lands immediately to the WAF, where there is configured a rule that recognizes the malicious attempt (the malicious payload pointed to by "`param`") and sends back a HTTP 404 error, in fact blocking the attack.

But what if we encode in base64 the URL

https://auth.targetdomain/vulnerable_endpoint?param=malicious_RCE_payload

which produces the following base64 string

"`aHR0cHM6Ly9hdXRoLnRhcmdldGRvbWFpbi92dWxuZXJhYmxlX2VuZHBvaW50P3BhcmFtPW1hbGljaW91c19SQ0VfcGF5bG9hZA==`"

and pass it to the "`cfru`" parameter as follows?

https//targetdomain/?**cfru**=aHR0cHM6Ly9hdXRoLnRhcmdldGRvbWFpbi92dWxuZXJhYmxlX2VuZHBvaW50P3BhcmFtPW1hbGljaW91c19SQ0VfcGF5bG9hZA==

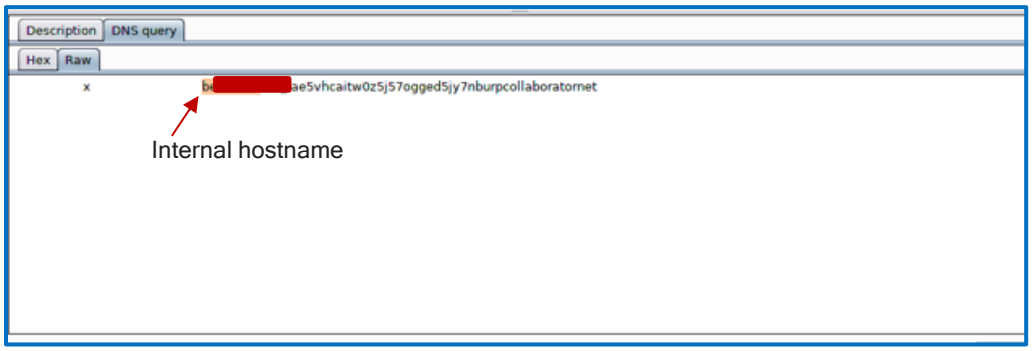
In our case:

1. The request crossed the WAF which had nothing to complain.

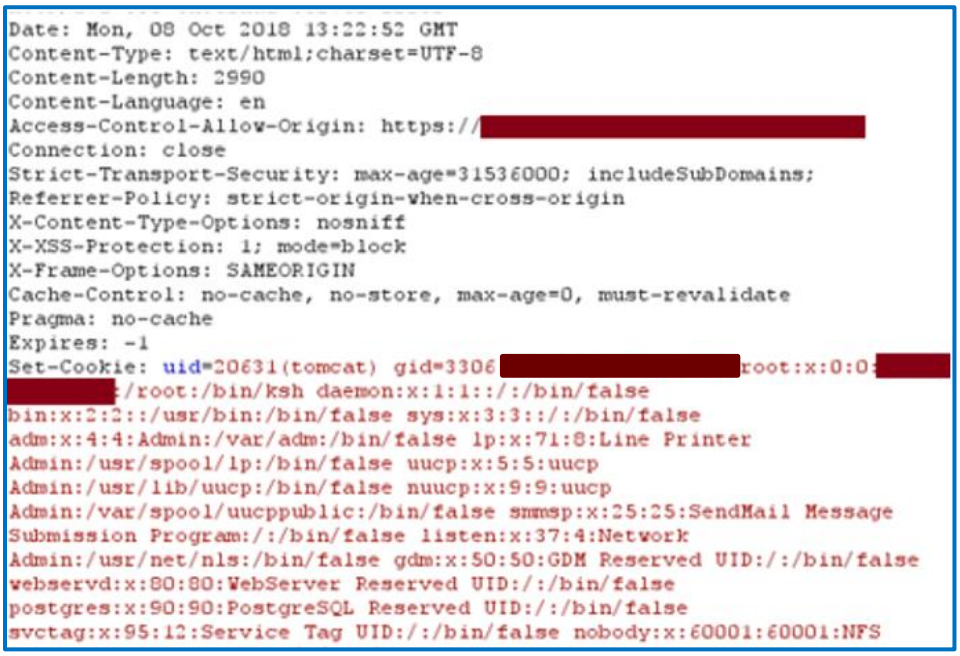
2. Then it arrived to the Bluecoat device that in turn base64-decoded the "`cfru`" parameter and issued a GET request toward the internal host https://auth.targetdomain/vulnerable_endpoint?param=malicious_RCE_payload.
3. This in turn triggered the vulnerability.

BINGO!

And bingo! We can see the output of our malicious payload (nothing more than the "`hostname`" command) exfiltrated via DNS (outgoing TCP connections to our host located in the internet were indeed blocked as already said previously).



Furthermore, we played a bit with our malicious payload in order to have the output of our injected commands returned directly as part of the HTTP headers in the server reply.



CONCLUSION

There are at least two mistakes that can be spot here:

- The bluecoat device was behaving as a request "forwarder" instead of responding with a HTTP redirect as happened for other URLs (that would have caused the subsequent client requests to be caught and blocked by WAF).

- There was no rule implemented at WAF level that base64-decoded the "`cfru`" parameter before passing it to the Bluecoat device, in order to analyze whether or not the request's content matched with one of the blocking rules deployed in the WAF itself.

Good for us! We notified the vulnerability to the vendor straight away and they decided to recognize us a nice bug bounty!

The bottom line here is that virtual patching is ok if you need a bit of extra time before fixing a serious vulnerability. But if you use it in place of real patching, well it is only question of time before you will get hacked.

If you want to know more about similar exploitation techniques and other web hacking tricks, check out our Blackhat Las Vegas courses on August 1-2[1] and 3-4[2] 2020, because this will be one of the topics covered there.

Twitter: https://twitter.com/redtimmysec
Blog: https://www.redtimmy.com/blog/

[1] https://www.blackhat.com/us-20/training/schedule/index.html#practical-web-application-hacking-advanced-18992

[2] https://www.blackhat.com/us-20/training/schedule/#practical-web-application-hacking-advanced-189921578438852