Phishing signatures creation HOWTO

Török Edwin

1 Database file format

1.1 PDB format

This file contains urls/hosts that are target of phishing attempts. It contains lines in the following format:

```
R[Filter]:RealURL:DisplayedURL[:FuncLevelSpec]
H[Filter]:DisplayedHostname[:FuncLevelSpec]
```

R regular expression, for the concatenated URL

H matches the `DisplayedHostname` as a simple pattern (literally, no regular expression)

- the pattern can match either the full hostname
- or a subdomain of the specified hostname
- to avoid false matches in case of subdomain matches, the engine checks that there is a dot(`.`) or a space( ) before the matched portion

Filter is ignored for R and H for compatibility reasons

REALURL is the URL the user is sent to, example: *href* attribute of an html anchor (<*a*> *tag*)

DISPLAYEDURL is the URL description displayed to the user, where its *claimed* they are sent, example: contents of an html anchor (<*a*> *tag*)

DisplayedHostname is the hostname portion of the DISPLAYEDURL

FuncLevelSpec an (optional) functionality level, 2 formats are possible:

- `minlevel` all engines having functionality level >= `minlevel` will load this line
- `minlevel-maxlevel` engines with functionality level >= `minlevel`, and < `maxlevel` will load this line

1.2 GDB format

This file contains URL hashes in the following format:

```
S:P:HostPrefix[:FuncLevelSpec]
S:F:Sha256hash[:FuncLevelSpec]
S1:P:HostPrefix[:FuncLevelSpec]
S1:F:Sha256hash[:FuncLevelSpec]
S2:P:HostPrefix[:FuncLevelSpec]
S2:F:Sha256hash[:FuncLevelSpec]
S:W:Sha256hash[:FuncLevelSpec]
```

S: These are hashes for Google Safe Browsing - malware sites, and should not be used for other purposes.

S2: These are hashes for Google Safe Browsing - phishing sites, and should not be used for other purposes.

S1: Hashes for blacklisting phishing sites. Virus name: Phishing.URL.Blacklisted

S:W Locally whitelisted hashes.

HostPrefix 4-byte prefix of the sha256 hash of the last 2 or 3 components of the hostname. If prefix doesn't match, no further lookups are performed.

Sha256hash sha256 hash of the canonicalized URL, or a sha256 hash of its prefix/suffix according to the Google Safe Browsing "Performing Lookups" rules. There should be a corresponding `:P:HostkeyPrefix` entry for the hash to be taken into consideration.

To see which hash/URL matched, look at the `clamscan --debug` output, and look for the following strings: `Looking up hash`, `prefix matched`, and `Hash matched`. Local whitelisting of .gdb entries can be done by creating a local.gdb file, and adding a line `S:W:<HASH>`.

1.3 WDB format

This file contains whitelisted url pairs It contains lines in the following format:

```
X:RealURL:DisplayedURL[:FuncLevelSpec]
M:RealHostname:DisplayedHostname[:FuncLevelSpec]
```

X regular expression, for the *entire URL*, not just the hostname

- The regular expression is by default anchored to start-of-line and end-of-line, as if you have used `^RegularExpression$`
- A trailing `/` is automatically added both to the regex, and the input string to avoid false matches
- The regular expression matches the *concatenation* of the REALURL, a colon(`:`), and the DISPLAYEDURL as a single string. It doesn't separately match REALURL and DISPLAYEDURL!

M matches hostname, or subdomain of it, see notes for H above

1.4 Hints

- empty lines are ignored
- the colons are mandatory
- Don't leave extra spaces on the end of a line!
- if any of the lines don't conform to this format, clamav will abort with a Malformed Database Error
- see section **??** for more details on REALURL/DISPLAYEDURL

1.5 Examples of PDB signatures

To check for phishing mails that target amazon.com, or subdomains of amazon.com:

```
H:amazon.com
```

To do the same, but for amazon.co.uk:

```
H:amazon.co.uk
```

To limit the signatures to certain engine versions:

```
H:amazon.co.uk:20-30
H:amazon.co.uk:20-
H:amazon.co.uk:0-20
```

First line: engine versions 20, 21, ..., 29 can load it
Second line: engine versions >= 20 can load it
Third line: engine versions < 20 can load it
In a real situation, you'd probably use the second form. A situation like that would be if you are using a feature of the signatures not available in earlier versions, or if earlier versions have bugs with your signature. Its neither case here, the above examples are for illustrative purposes only.

1.6 Examples of WDB signatures

To allow amazon's country specific domains and amazon.com, to mix domain names in DISPLAYEDURL, and REALURL:

```
X:.+\.amazon\.(at|ca|co\.uk|co\.jp|de|fr)([/?].*)?:.+\.amazon\.com([/?].*)?:17-
```

Explanation of this signature:

X: this is a regular expression

:17- load signature only for engines with functionality level >= 17 (recommended for type X)

The regular expression is the following (X:, :17- stripped, and a / appended)

```
.+\.amazon\.(at|ca|co\.uk|co\.jp|de|fr)([/?].*)?:.+\.amazon\.com([/?].*)?/
```

Explanation of this regular expression (note that it is a single regular expression, and not 2 regular expressions splitted at the :).

- `.+` any subdomain of
- `\.amazon\.` domain we are whitelisting (REALURL part)
- `(at|ca|co\.uk|co\.jp|de|fr)` country-domains: at, ca, co.uk, co.jp, de, fr
- `([/?].*)?` recomended way to end real url part of whitelist, this protects against embedded URLs (evilurl.example.com/amazon.co.uk/)
- `:` REALURL and DISPLAYEDURL are concatenated via a :, so match a literal : here
- `.+` any subdomain of
- `\.amazon\.com` whitelisted DisplayedURL
- `([/?].*)?` recommended way to end displayed url part, to protect against embedded URLs
- `/` automatically added to further protect against embedded URLs

When you whitelist an entry make sure you check that both domains are owned by the same entity. What this whitelist entry allows is: Links claiming to point to amazon.com (DISPLAYEDURL), but really go to country-specific domain of amazon (REALURL).

1.7 Example for how the URL extractor works

Consider the following HTML file:

```
<html>
<a href="http://1.realurl.example.com/">
  1.displayedurl.example.com
</a>
<a href="http://2.realurl.example.com">
  2 d<b>i<p>splayedurl.e</b>xa<i>mple.com
</a>
<a href="http://3.realurl.example.com">
  3.nested.example.com
  <a href="http://4.realurl.example.com">
    4.displayedurl.example.com
  </a>
</a>
<form action="http://5.realurl.example.com">
  sometext
  <img src="http://5.displayedurl.example.com/img0.gif"/>
  <a href="http://5.form.nested.displayedurl.example.com">
    5.form.nested.link-displayedurl.example.com
  </a>
</form>
<a href="http://6.realurl.example.com">
  6.displ
  <img src="6.displayedurl.example.com/img1.gif"/>
```

```
  ayedurl.example.com
</a>
<a href="http://7.realurl.example.com">
  <iframe src="http://7.displayedurl.example.com">
</a>
```

The phishing engine extract the following REALURL/DISPLAYEDURL pairs from it:

```
http://1.realurl.example.com/
1.displayedurl.example.com

http://2.realurl.example.com
2displayedurl.example.com

http://3.realurl.example.com
3.nested.example.com

http://4.realurl.example.com
4.displayedurl.example.com

http://5.realurl.example.com
http://5.displayedurl.example.com/img0.gif

http://5.realurl.example.com
http://5.form.nested.displayedurl.example.com

http://5.form.nested.displayedurl.example.com
5.form.nested.link-displayedurl.example.com

http://6.realurl.example.com
6.displayedurl.example.com

http://6.realurl.example.com
6.displayedurl.example.com/img1.gif
```

1.8 How matching works

1.8.1 RealURL, displayedURL concatenation

The phishing detection module processes pairs of REALURL/DISPLAYEDURL. Matching against daily.wdb is done as follows: the REALURL is concatenated with a `:`, and with the DISPLAYEDURL, then that *line* is matched against the lines in daily.wdb/daily.pdb

So if you have this line in daily.wdb:

```
M:www.google.ro:www.google.com
```

and this href: `<a href='http://www.google.ro'>www.google.com</a>` then it will be whitelisted, but: `<a href='http://images.google.com'>www.google.com</a>` will not.

1.8.2 What happens when a match is found

In the case of the whitelist, a match means that the REALURL/DISPLAYEDURL combination is considered CLEAN, and no further checks are performed on it.

In the case of the domainlist, a match means that the REALURL/DISPLAYEDURL is going to be checked for phishing attempts.

Furthermore you can restrict what checks are to be performed by specifying the 3-digit hexnumber.

1.8.3 Extraction of REALURL, DISPLAYEDURL from HTML tags

The html parser extracts pairs of REALURL/DISPLAYEDURL based on the following rules.

In version 0.93: After URLs have been extracted, they are normalized, and cut after the hostname. `http://test.example.com/path/somecgi?queryparameters` becomes `http://test.example.com/`

a (anchor) the *href* is the REALURL, its *contents* is the DISPLAYEDURL

contents is the tag-stripped contents of the <a> tags, so for example <b> tags are stripped (but not their contents)

nesting another <a> tag withing an <a> tag (besides being invalid html) is treated as a <a..

form the *action* attribute is the REALURL, and a nested <a> tag is the DISPLAYEDURL

img/area if nested within an <*a*> tag, the REALURL is the *href* of the a tag, and the *src/dynsrc/area* is the DISPLAYEDURL of the img

if nested withing a *form* tag, then the action attribute of the *form* tag is the REALURL

iframe if nested withing an <*a*> tag the *src* attribute is the DISPLAYEDURL, and the *href* of its parent *a* tag is the REALURL

if nested withing a *form* tag, then the action attribute of the *form* tag is the REALURL

1.8.4 Example

Consider this html file:

www.paypal.com

click here to sign in

<form action="evilurl_form">

Please sign in to Ebay using this form

<input type='text' name='username'>Username</input>

....

</form>

<img src="images.paypal.com/secure.jpg">

The resulting REALURL/DISPLAYEDURL pairs will be (note that one tag can generate multiple pairs):

- evilurl / www.paypal.com
- evilurl2 / click here to sign in
- evilurl2 / www.ebay.com
- evilurl_form / cgi.ebay.com
- cgi.ebay.com / Ebay
- evilurl / image.paypal.com/secure.jpg

1.9 Simple patterns

Simple patterns are matched literally, i.e. if you say:

    www.google.com

it is going to match *www.google.com*, and only that. The . *(dot)* character has no special meaning (see the section on regexes **??** for how the *.(dot)* character behaves there)

1.10 Regular expressions

POSIX regular expressions are supported, and you can consider that internally it is wrapped by ^, and *$*. In other words, this means that the regular expression has to match the entire concatenated (see section **??** for details on concatenation) url.

It is recomended that you read section **??** to learn how to write regular expressions, and then come back and read this for hints.

Be advised that clamav contains an internal, very basic regex matcher to reduce the load on the regex matching core. Thus it is recomended that you avoid using regex syntax not supported by it at the very beginning of regexes (at least the first few characters).

Currently the clamav regex matcher supports:

- . (dot) character
- \ (escaping special characters)
- | (pipe) alternatives
- [] (character classes)
- () (parenthesis for grouping, but no group extraction is performed)
- other non-special characters

Thus the following are not supported:

- + repetition
- * repetition

- { } repetition
- backreferences
- lookaround
- other “advanced” features not listed in the supported list ;)

This however shouldn’t discourage you from using the “not directly supported features “, because if the internal engine encounters unsupported syntax, it passes it on to the POSIX regex core (beginning from the first unsupported token, everything before that is still processed by the internal matcher). An example might make this more clear:

www\.google\.(com|ro|it) ([a-zA-Z])+\.google\.(com|ro|it)

Everything till *([a-zA-Z])+* is processed internally, that parenthesis (and everything beyond) is processed by the posix core.

Examples of url pairs that match:

- *www.google.ro images.google.ro*
- www.google.com images.google.ro

Example of url pairs that don’t match:

- www.google.ro images1.google.ro
- images.google.com image.google.com

1.11 Flags

Flags are a binary OR of the following numbers:

HOST_SUFFICIENT 1

DOMAIN_SUFFICIENT 2

DO_REVERSE_LOOKUP 4

CHECK_REDIR 8

CHECK_SSL 16

CHECK_CLOAKING 32

CLEANUP_URL 64

CHECK_DOMAIN_REVERSE 128

CHECK_IMG_URL 256

DOMAINLIST_REQUIRED 512

The names of the constants are self-explanatory.

These constants are defined in libclamav/phishcheck.h, you can check there for the latest flags.

There is a default set of flags that are enabled, these are currently:

(CLEANUP_URL|CHECK_SSL|CHECK_CLOAKING|CHECK_IMG_URL)

ssl checking is performed only for a tags currently.

You must decide for each line in the domainlist if you want to filter any flags (that is you don't want certain checks to be done), and then calculate the binary OR of those constants, and then convert it into a 3-digit hexnumber. For example you devide that domain_sufficient shouldn't be used for ebay.com, and you don't want to check images either, so you come up with this flag number: $2|256 \Rightarrow 258(decimal) \Rightarrow 102(hexadecimal)$

So you add this line to daily.wdb:

- R102 www.ebay.com .+

2 Introduction to regular expressions

Recomended reading:

- http://www.regular-expressions.info/quickstart.html
- http://www.regular-expressions.info/tutorial.html
- regex(7) man-page: http://www.tin.org/bin/man.cgi?section=7&topic=regex

2.1 Special characters

[the opening square bracket - it marks the beginning of a character class, see section **??**

\ the backslash - escapes special characters, see section **??**

^ the caret - matches the beginning of a line (not needed in clamav regexes, this is implied)

$ the dollar sign - matches the end of a line (not needed in clamav regexes, this is implied)

. the period or dot - matches *any* character

| the vertical bar or pipe symbol - matches either of the token on its left and right side, see section **??**

? the question mark - matches optionally the left-side token, see section **??**

***** the asterisk or star - matches 0 or more occurences of the left-side token, see section ??

+ the plus sign - matches 1 or more occurences of the left-side token, see section ??

(the opening round bracket - marks beginning of a group, see section **??**

) the closing round bracket - marks end of a group, see section ??

2.2 Character classes

2.3 Escaping

Escaping has two purposes:

- it allows you to actually match the special characters themselves, for example to match the literal +, you would write \+
- it also allows you to match non-printable characters, such as the tab (\t), newline (\n), ..

However since non-printable characters are not valid inside an url, you won't have a reason to use them.

2.4 Alternation

2.5 Optional matching, and repetition

2.6 Groups

Groups are usually used together with repetition, or alternation. For example: *(com|it)+* means: match 1 or more repetitions of *com* or *it,* that is it matches: com, it, comcom, comcomcom, comit, itit, ititcom,... you get the idea.

Groups can also be used to extract substring, but this is not supported by the clam engine, and not needed either in this case.

3 How to create database files

3.1 How to create and maintain the whitelist (daily.wdb)

If the phishing code claims that a certain mail is phishing, but its not, you have 2 choices:

- examine your rules daily.pdb, and fix them if necessary (see: section **??**)
- add it to the whitelist (discussed here)

Lets assume you are having problems because of links like this in a mail:

```
<a href=''http://69.0.241.57/bCentral/L.asp?L=XXXXXXXX''>
  http://www.bcentral.it/
</a>
```

After investigating those sites further, you decide they are no threat, and create a line like this in daily.wdb:

R http://www\.bcentral\.it/.+ http://69\.0\.241\.57/bCentral/L\.asp?L=.+

Note: urls like the above can be used to track unique mail recipients, and thus know if somebody actually reads mails (so they can send more spam). However since this site required no authentication information, it is safe from a phishing point of view.

3.2 How to create and maintain the domainlist (daily.pdb)

When not using –phish-scan-alldomains (production environments for example), you need to decide which urls you are going to check.

Although at a first glance it might seem a good idea to check everything, it would produce false positives. Particularly newsletters, ads, etc. are likely to use URLs that look like phishing attempts.

Lets assume that you've recently seen many phishing attempts claiming they come from Paypal. Thus you need to add paypal to daily.pdb:

```
R .+ .+\.paypal\.com
```

The above line will block (detect as phishing) mails that contain urls that claim to lead to paypal, but they don't in fact.

Be carefull not to create regexes that match a too broad range of urls though.

3.3 Dealing with false positives, and undetected phishing mails

3.3.1 False positives

Whenever you see a false positive (mail that is detected as phishing, but its not), you need to examine *why* clamav decided that its phishing. You can do this easily by building clamav with debugging (./configure –enable-experimental –enable-debug), and then running a tool:

```
$contrib/phishing/why.py phishing.eml
```

This will show the url that triggers the phish verdict, and a reason why that url is considered phishing attempt.

Once you know the reason, you might need to modify daily.pdb (if one of yours rules inthere are too broad), or you need to add the url to daily.wdb. If you think the algorithm is incorrect, please file a bugreport on bugzilla.clamav.net, including the output of *why.py*.

3.3.2 Undetected phish mails

Using why.py doesn't help here unfortunately (it will say: clean), so all you can do is:

```
$clamscan/clamscan –phish-scan-alldomains undetected.eml
```

And see if the mail is detected, if yes, then you need to add an appropriate line to daily.pdb (see section **??**).

If the mail is not detected, then try using:

```
$clamscan/clamscan –debug undetected.emllless
```

Then see what urls are being checked, see if any of them is in a whitelist, see if all urls are detected, etc.