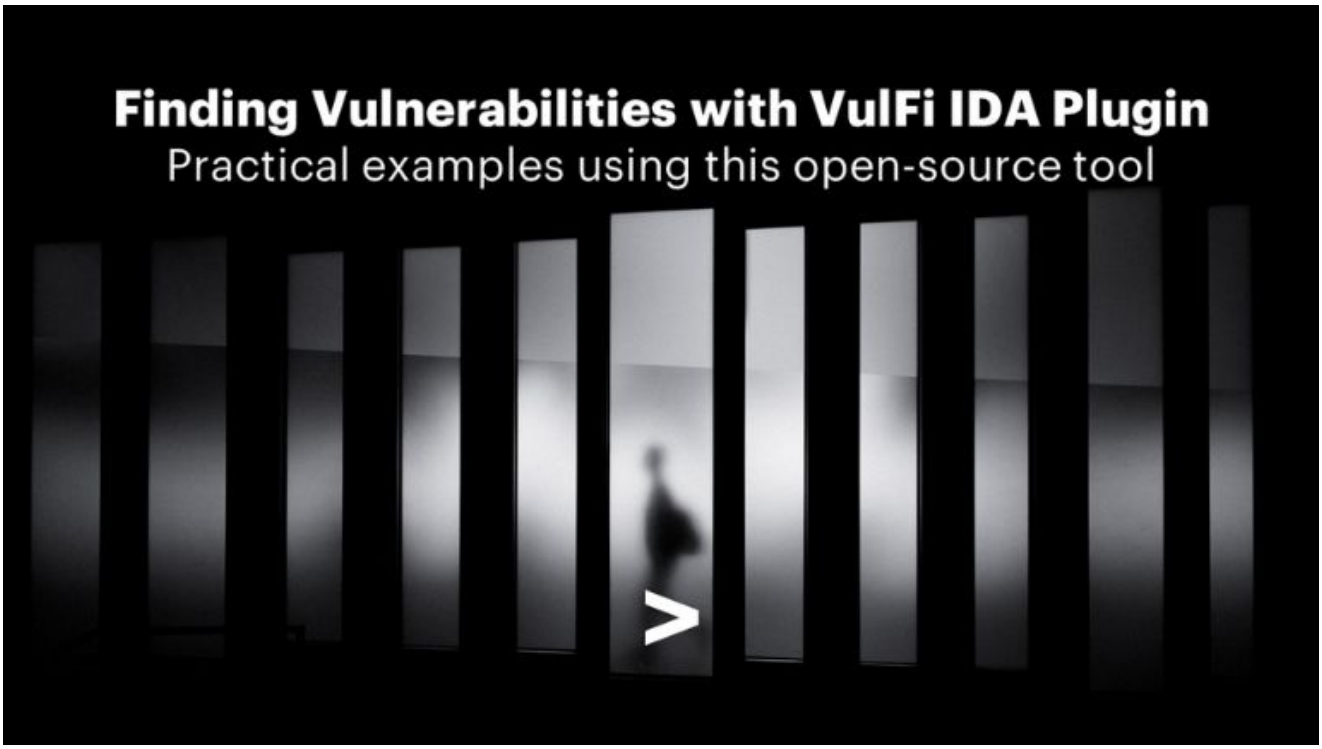
Finding Vulnerabilities with VulFi IDA Plugin

accenture.com/us-en/blogs/security/finding-vulnerabilities-vulfi-ida-plugin



Share

In March, we published an IDA Pro plugin that Accenture Security teams use to find vulnerabilities and other potentially interesting issues in the compiled binaries. The plugin provides a Python-based query language with which users can look for calls to specific functions that match criteria specified in the query. In this article, we will look at the high-level theory behind this tool and demonstrate its use on a practical example of finding vulnerabilities identified as *CVE-2022-26413* and *CVE-2022-26414*.

How the plugin works

When doing vulnerability research, it is quite common to look for a call to certain functions. And while cross-references shown by IDA are a good starting point, the idea for this plugin came from the need to filter thousands of uninteresting calls to a function and find only those that might be valuable from the security perspective.

To give a very generic example, imagine a binary file that calls a function like *strcpy* a thousand times. Out of all these occurrences, all use a static string as a second parameter, with only 50 exceptions. Without the way of filtering the function calls based on the

properties of the parameters that are passed to them (and their return value), the analyst would have to investigate all 1,000 cross-references. The worst part about this is that most of them would have to be dismissed as uninteresting due to the use of static values in the second argument.

This is the kind of case that's perfect for a plugin developed using the IDAPython API. The goals for the plugin are quite easy to define. We want an architecture-agnostic way of filtering function calls based on the properties of the parameters and returned values. The property could be, for example, whether the parameter is a constant value. In that case, we also want a way to check for specific constant values.

IDA offers a plethora of functions for processing disassembly as well as decompiler output. In cases where the decompiler could be used, the plugin will work much better, because the Hex-Rays processing that happens under the hood allows the VulFi plugin to access much more accurate values for function call parameters. For the cases where the disassembly is the only option, the task is a bit harder. If possible, the VulFi will try to apply function type for all known functions as defined in this file prior to running the search. With this, it will leverage the possibility to locate the assembly instruction that is responsible for loading the parameter and try to deduce its value from it. In case that the type-system is not supported for the architecture, the VulFi will just mark all the cross-references for the function and put them in the table.

With the search concluded, the results are placed in VulFi view. Since the plugin was developed with an assumption that search results will likely be numerous, a simple tracking and commenting feature was added to the plugin and will be demonstrated below in a practical walkthrough of the usage.

An example usage of the VulFi plugin

1. Finding the right target

For the practical example, I will use a firmware of the Zyxel VMG3312-T20A router that I happen to have in my drawer. The manufacturer announced some time ago that this model had reached the end of its life. Nonetheless, according to internal validations performed by Zyxel, the discovered vulnerabilities also affect several products that are still supported, as mentioned here.

The firmware for the router could be downloaded from here. With the firmware image downloaded, we can inspect its content. As shown below, the most interesting file is *V530ABFX5C0.bin* (mainly because of its size, but also because of the filename extension).

```
[zyxel]$> ls -lah
total 18M
drwxr-xr-x  2 mpet mpet 4.0K Feb 17 11:31  .
drwxr-xr-x 20 mpet mpet 4.0K Feb 17 11:30  ..
-rw-r--r--  1 mpet mpet  17M Jul 17  2019  V530ABFX5C0.bin
-rw-r--r--  1 mpet mpet 612K Apr  8  2020  V530ABFX5C0.pdf
-rw-r--r--  1 mpet mpet  79K Jun 26  2019  V530ABFX5C0.rom
-rw-r--r--_ 1 mpet mpet 603K Apr  7  2020 'VMG3312-T20A_V5.30(ABFX.5)C0-foss.pdf'
```

The *V530ABFX5C0.bin* file can be easily processed using a *binwalk* utility. This will successfully detect and extract a SquashFS file system.

```
[zyxel]$> binwalk V530ABFX5C0.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
372           0x174           LZMA compressed data, properties: 0x6C, dictionary
 size: 8388608 bytes, uncompressed size: 4660320 bytes
1508549       0x1704C5        Squashfs filesystem, little endian, version 4.0, c
ompression:lzma, size: 15670418 bytes, 1436 inodes, blocksize: 262144 bytes, cre
ated: 2019-06-26 07:48:54
```

The extracted contents of the file system probably contain many interesting files, however, since we know that the router in question has a feature-packed web interface, the best place to try the plugin would be the file */bin/zhttpd*. This file implements the logic of handling the requests coming from the user browser and thus provides a convenient way for us to test any potential issues.

```
[squashfs-root]$> ls -lah bin/zhttpd
-rwxr-xr-x 1 mpet mpet 338K Jun 26  2019 bin/zhttpd
[squashfs-root]$>
```

2. Initial peek at the binary

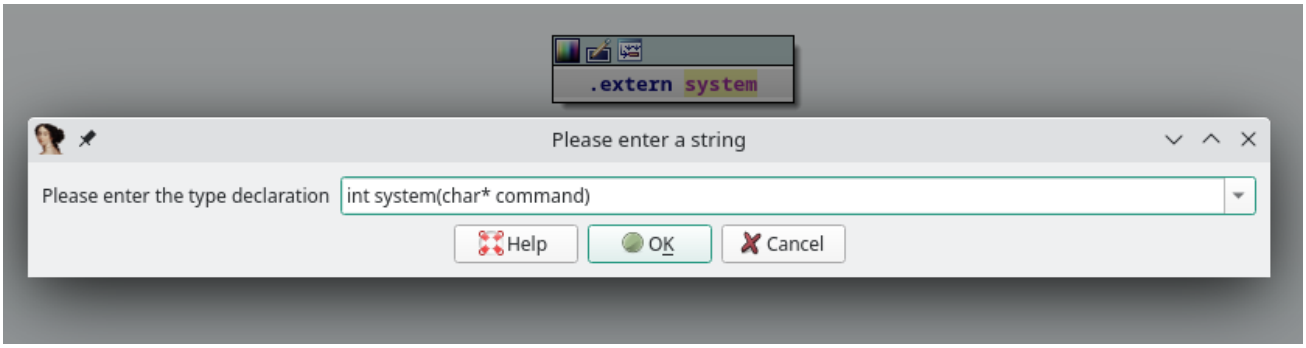
The initial analysis of the binary starts obviously by loading it in the IDA Pro. After the analysis is completed, we can see that the binary is an ELF file for a 32-bit big-endian MIPS architecture.

```
LOAD:00400000  # File Name   : /home/mpet/Downloads/zyxel/_V530ABFX5C0.bin.extracted/squashfs-root/bin/zhttpd
LOAD:00400000  # Format      : ELF for MIPS (Executable)
LOAD:00400000  # Imagebase   : 400000
LOAD:00400000  # Interpreter '/lib/ld-uClibc.so.0'
LOAD:00400000  # Needed Library 'libpthread.so.0'
LOAD:00400000  # Needed Library 'libclinkc.so'
LOAD:00400000  # Needed Library 'libexpat.so.0'
LOAD:00400000  # Needed Library 'libssl.so.1.0.0'
LOAD:00400000  # Needed Library 'libcrypto.so.1.0.0'
LOAD:00400000  # Needed Library 'libuuid.so.1'
LOAD:00400000  # Needed Library 'libzcfg_fe_rdm_access.so'
LOAD:00400000  # Needed Library 'libzcmd_tool.so'
LOAD:00400000  # Needed Library 'libjson-c.so.2'
LOAD:00400000  # Needed Library 'libzcfg_msg.so'
LOAD:00400000  # Needed Library 'libzcfg_fe_schema.so'
LOAD:00400000  # Needed Library 'libzcfg_fe_rdm_string.so'
LOAD:00400000  # Needed Library 'libzcfg_fe_rdm_struct.so'
LOAD:00400000  # Needed Library 'libzywwan.so'
LOAD:00400000  # Needed Library 'libzyutil.so'
LOAD:00400000  # Needed Library 'libzcfg_fe_dal.so'
LOAD:00400000  # Needed Library 'libzlog.so'
LOAD:00400000  # Needed Library 'libgcc_s.so.1'
LOAD:00400000  # Needed Library 'libc.so.0'
LOAD:00400000  #
LOAD:00400000  # Options     : --opsex
LOAD:00400000  # Options     : -fPIC
LOAD:00400000  # Options     : -fCPIC
LOAD:00400000  # Options     : -mips32r2
LOAD:00400000  # Options     : -mabi=32
```

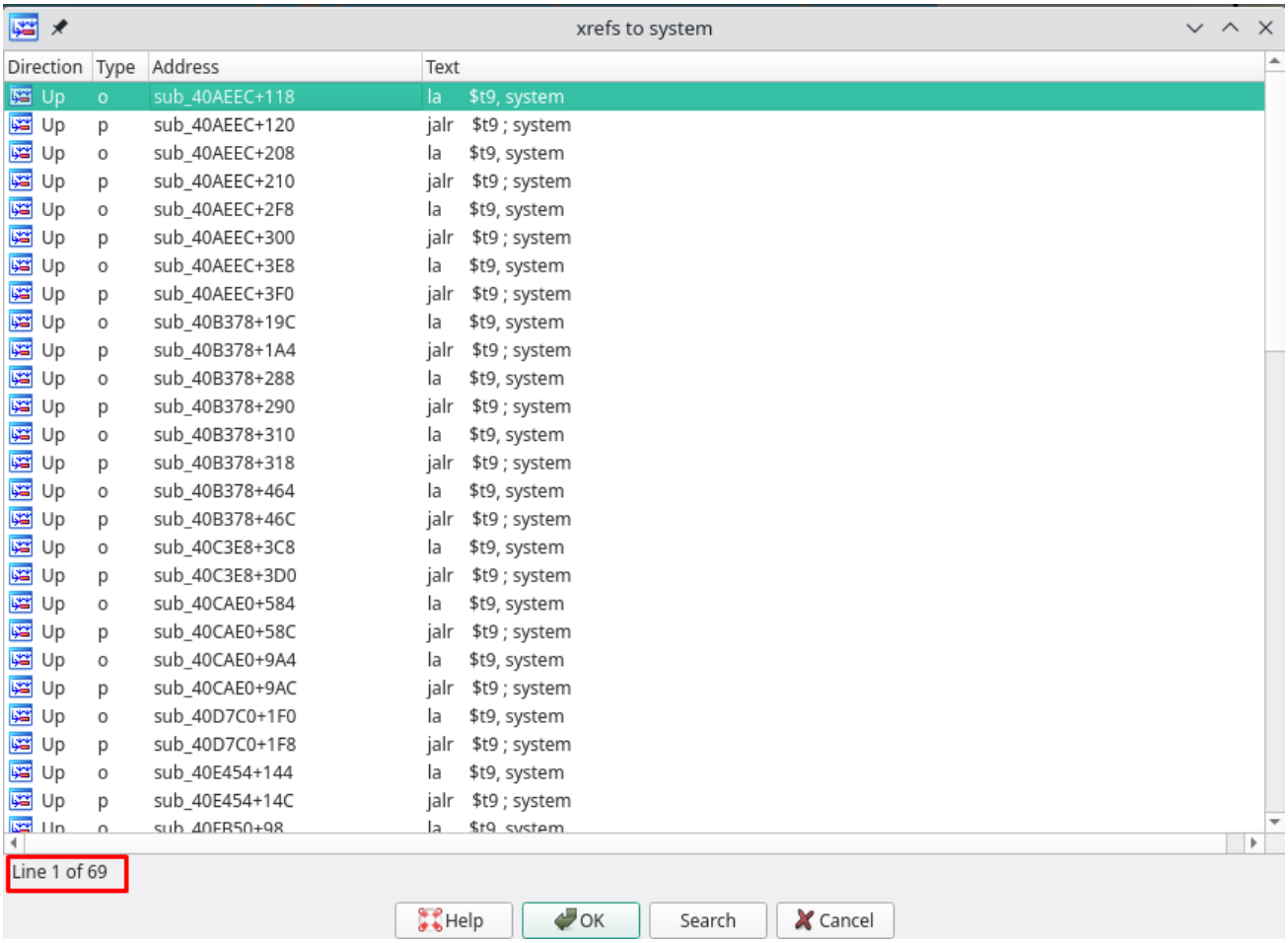
After looking around the used functions, we can see that the binary is using function *system*, which is used for executing OS commands.

```
extern:0046490C                 .extern system          # CODE XREF: sub_40AEEC+120↑p
extern:0046490C                                         # sub_40AEEC+210↑p ...
```

To make life for VulFi easier, we must set the function type according to the official documentation (the dialog for type configuration can be invoked by pressing *Y*).

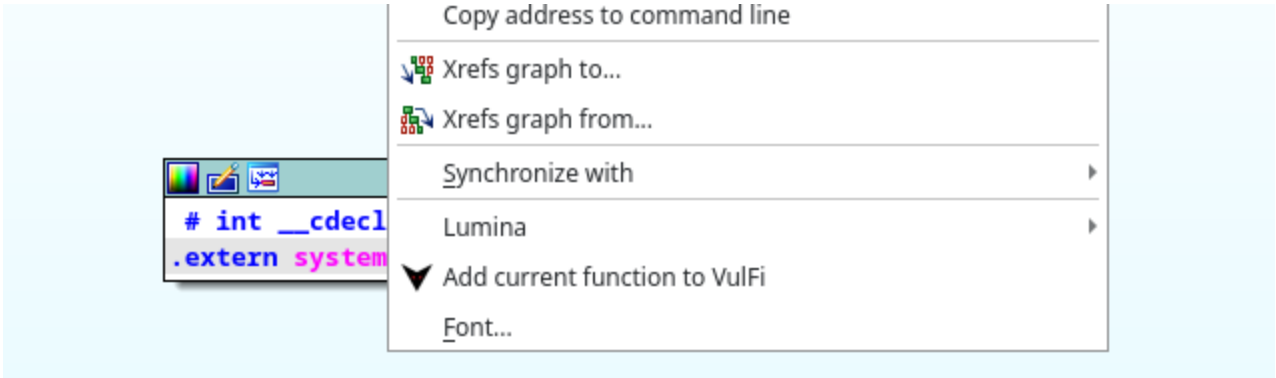


We can also check the current count of the cross-references to this function. As shown below, this binary contains a total of 69 unique calls to function *system*.

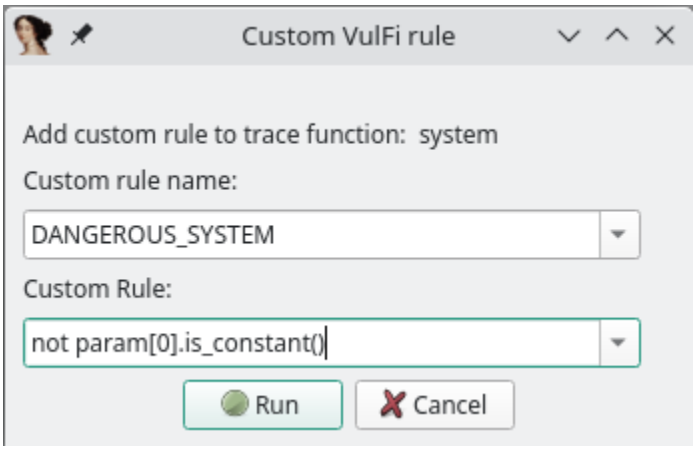


3. Using VulFi

Let's see if VulFi can save us some time by only showing us those calls in which the first and only argument of the *system* function is set to a non-static value. To find out, we must set a custom rule that will look for such occasions (this rule is also in the default set, however, for the sake of the article, let us recreate it). To initiate a setup of the new rule, set IDA view to the body of the function that you want to look for, right-click anywhere in the body (in this case we right-click the *system* label) and select the option "Add current function to VulFi".

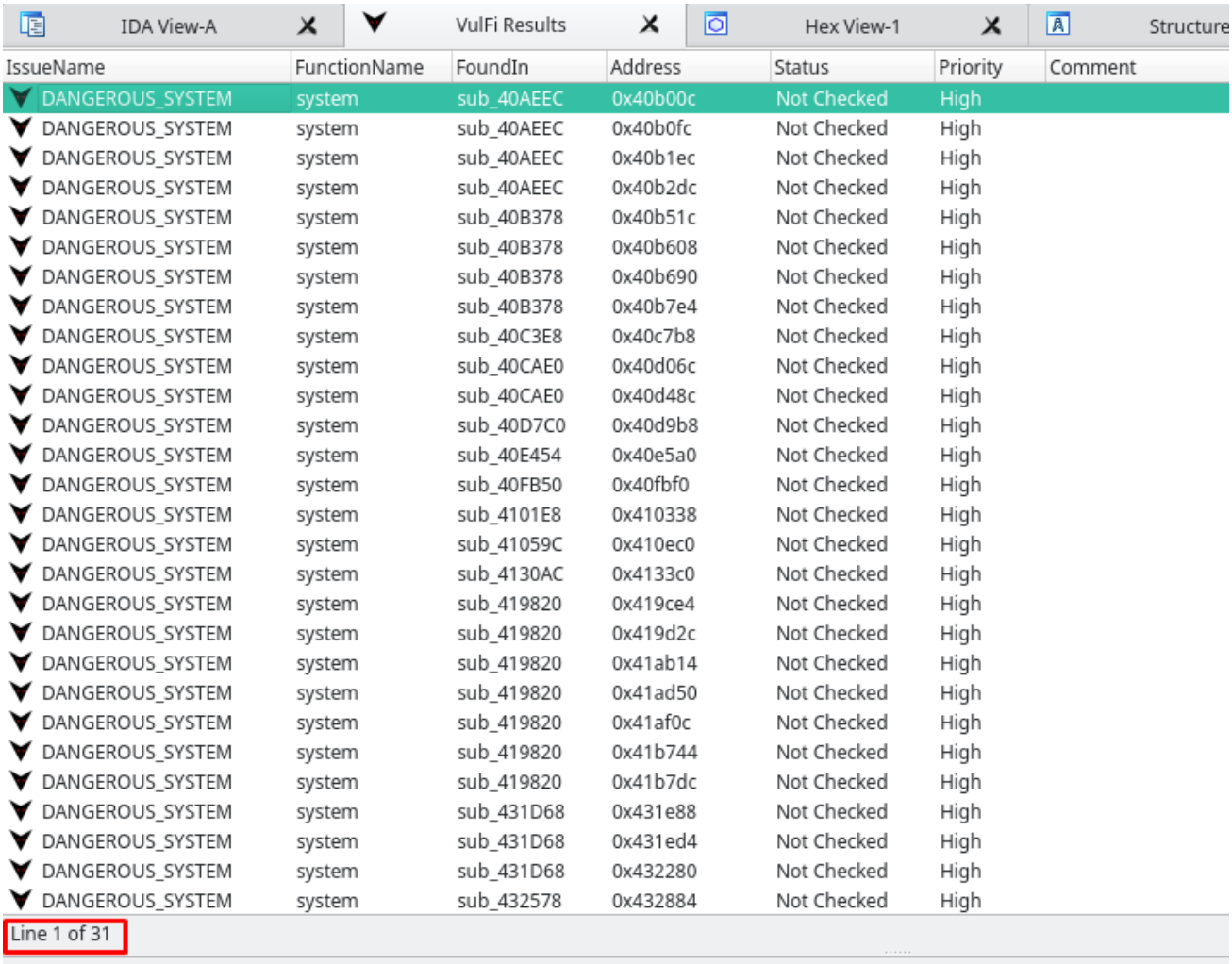


Selecting this option will spawn a simple dialog with two required fields. The first field is the name of the new custom rule so that you can easily find it amongst other results that might already be in the result list. The second field is where the magic happens; that is where you specify the rule. Since we are looking for any occurrence of the call to *system* function where the first parameter is not constant value the rule will have a form as shown in the screenshot below:



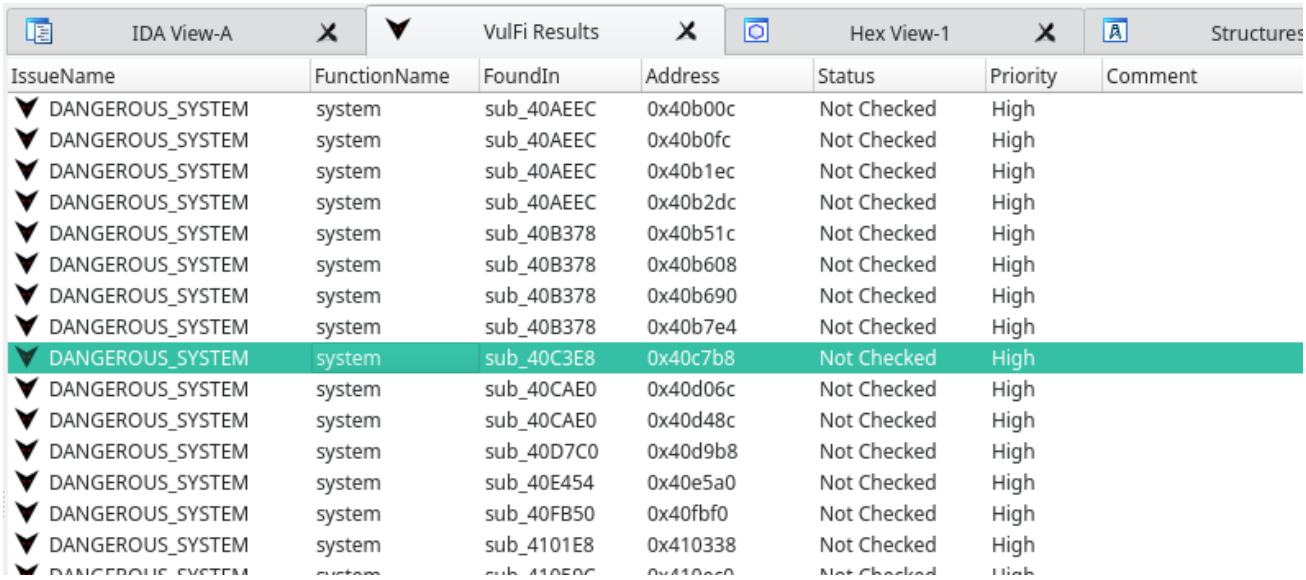
A brief description of the above rule is likely required at this point. We start with the *not* keyword to negate the expression. We are looking for the first parameter, that is why we use an array of parameters called *param* and we use the first item in the list (*[0]*). The state of the parameter that we are interested in is whether it is a constant. This can be achieved by calling a function *is_constant()* on the parameter object, the negation which we put in the beginning will make sure that we only get results where the *is_constant()* function returned *False*. As you may have noticed, the syntax is very similar to conditions as written in Python. In fact, this is a Python code, it is just that several functions have been prepared for you to build a sort-of query language. If you would like to find out more about available functions, please see the README file in the official repository of this plugin.

Let us get back to the example now. When you press the Run button, VulFi will see if the decompiler for the given architecture is available and if it is, it will automatically use it. Therefore, you will see progress pop-ups mostly linked to the decompiler processing the functions. After the process of searching is completed, you will be presented with VulFi results view. In the case of the *zhttpd* binary and the search for the rule defined above, we can see that thanks to VulFi, we are left with only 31 out of the original 69 cross-references.



4. Inspecting a vulnerable code (CVE-2022-26413)

To answer the question in the subtitle for this section, we can just look at the VulFi results. Amongst all the detected calls to *system* function let's have a look at function *sub_40C3E8*. This can be easily done by double-clicking the line with this function in VulFi, this will automatically make the main IDA view switch to the location where the call was identified.



IDA View-A | VulFi Results | Hex View-1 | Structures

IssueName	FunctionName	FoundIn	Address	Status	Priority	Comment
DANGEROUS_SYSTEM	system	sub_40AEEC	0x40b00c	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40AEEC	0x40b0fc	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40AEEC	0x40b1ec	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40AEEC	0x40b2dc	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40B378	0x40b51c	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40B378	0x40b608	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40B378	0x40b690	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40B378	0x40b7e4	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40C3E8	0x40c7b8	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40CAE0	0x40d06c	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40CAE0	0x40d48c	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40D7C0	0x40d9b8	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40E454	0x40e5a0	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_40FB50	0x40fbf0	Not Checked	High	
DANGEROUS_SYSTEM	system	sub_4101E8	0x410338	Not Checked	High	

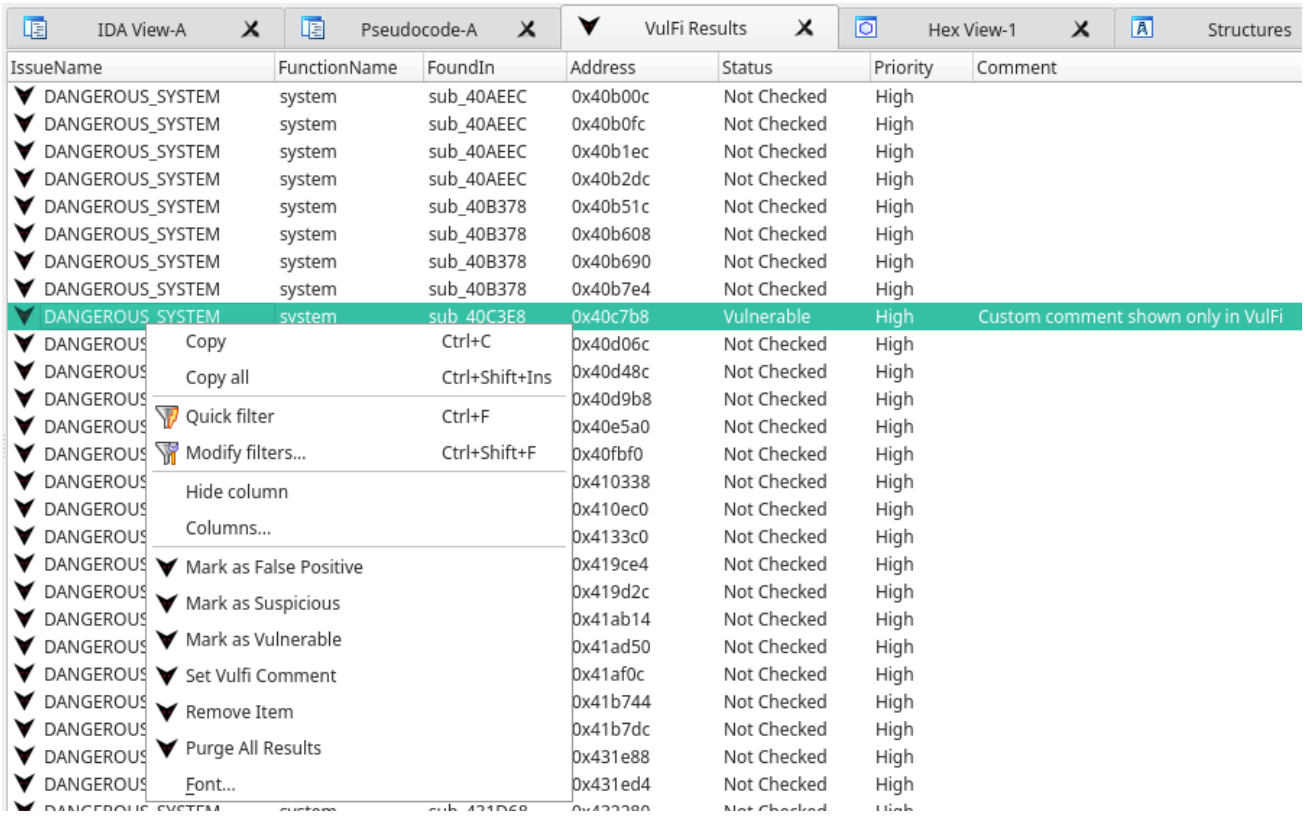
Please note that for the sake of better readability, the remainder of this article uses the decompiler in IDA. As you can see below, the marked call to *system* function does indeed accept dynamic argument.

IDA View-A | Pseudocode-A | VulFi Results | Hex View-1

```
66  v6 = cg_filelist_getBy_valname(*(_DWORD *)(a1 + 668), "certImportFileName");
67  if ( v6 )
68  {
69    v5 = (const char *)cg_file_getname(v6);
70    if ( v5 )
71    {
72      sprintf(v18, "mv %s %s", v5, "/var/local_cert");
73      printf("cmd = %s\n", v18);
74      system(v18);
75      v18[0] = 0;
76    }
77    else
78    {
79      puts("Certificate Import: Cannot get filepath....");
80      v3 = 0;
81    }
82  }
83  else
```

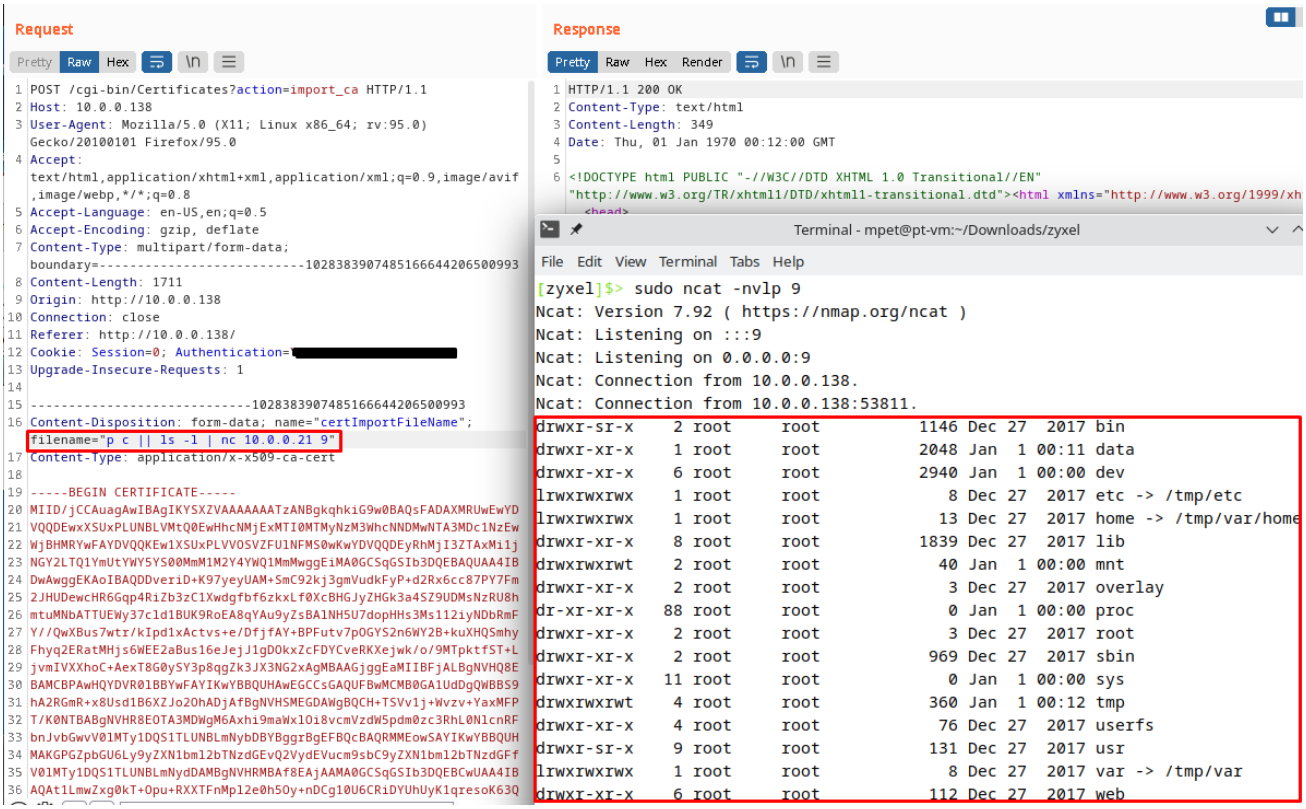
The vulnerability occurs on line 74 in the above snippet. To reach to that code, you must invoke action *import_ca* (not shown in here). This is done by sending a multipart request with the CA file in the parameter called *certImportFileName*. As can be deduced from the code on line 69, the name of the file sent in the multipart request will be used in the *sprintf* (CVE-2022-26414) function to build a command string (line 72) that is passed to the *system* function on line 74.

Since we have identified a place that is most likely vulnerable, we can go back to VulFi view and use a right click on the given item to either set a custom comment or to set a status for the item to one of the available options (False Positive, Suspicious or Vulnerable). This feature was added to make tracking of the progress easier as it is assumed that larger binaries will take multiple days to process.



5. Exploitation

Finally, we should prove the exploitability of the issue that we just found. That requires capturing a request in the intercepting proxy of our choice (BurpSuite is used in the example) and sending it with a modified filename parameter. The value set in this parameter in the below screen capture instructed the router to execute the *ls -l* command and pass the result of it to the attacker machine via *nc* connection. As can be seen by the highlighted sections, this was successful and thus a possibility to inject OS commands was proven.



Vulnerability Disclosure Process

The following dates are an important milestone related to the discovered vulnerabilities.

- 13 January 2022 – Issues reported to Zyxel
- 16 January 2022 – Vulnerabilities were acknowledged to be existent in the End-of-Life product
- 12 April 2022 – Advisory published by Zyxel (https://www.zyxel.com/support/OS-command-injection-and-buffer-overflow-vulnerabilities-of-CPE-and-ONTs.shtml)

Accenture Security *is a leading provider of end-to-end cybersecurity services, including advanced cyber defense, applied cybersecurity solutions and managed security operations. We bring security innovation, coupled with global scale and a worldwide delivery capability through our network of Advanced Technology and Intelligent Operations centers. Helped by our team of highly skilled professionals, we enable clients to innovate safely, build cyber resilience and grow with confidence. Follow us* @AccentureSecure *on Twitter,* LinkedIn *or visit us at* accenture.com/security*.*

do not constitute legal advice, nor do they necessarily reflect the views of Accenture, its subsidiaries, or affiliates. Given the inherent nature of threat intelligence, the content contained in this article is based on information gathered and understood at the time of its creation. It is subject to change. Accenture provides the information on an “as-is” basis without representation or warranty and accepts no liability for any action or failure to act taken in response to the information contained or referenced in this report.

Martin Petran

Associate Manager – Technology

Martin is member of the Embedded Systems Security Assessment team specializing in reverse engineering and firmware exploitation.