Natch: Detecting Attack Surface for Multi-Service Systems with Hybrid Introspection

Pavel Dovgalyuk, Maria Klimushenkova, Natalia Fursova, Ivan Vasiliev, and Vladislav Stepanov
Institute for System Programming, Moscow, Russia
{pavel.dovgalyuk, maria.klimushenkova, natalia.fursova, ivan.vasiliev, vladislav.stepanov}@ispras.ru

Abstract—Attack surface is the set of code and data that can be changed, stolen, or exploited by the user of the system. Attack surface of the complex systems may consist of functions on different programming languages and data files within web servers, Python/JS backends, databases, firewalls, and other services.

Having the attack surface description, one can remove unneeded dependencies of the scripts, modules, and programs, or improve testing (e.g., make more fuzzing), as it is needed for security development lifecycle (SDL). Attack surface description may also be required in the process of software certification.

In this paper we focus on finding program code that processes the input data and therefore can be treated as an attack surface. This analysis is performed in the virtual machine (VM), and application properties can be extracted using VM introspection. Existing introspection methods can be applied only to binary code, and not for scripts. When script interpreter runs a program, this script, and its functions should also be treated as an attack surface.

Runtime architecture of the system, i.e., loaded binary modules, executed scripts, open web sockets, accessed files, application interactions, and so on, also may be used for analysis of the attack surface.

Therefore the aim of this paper is extending introspection and VM analysis methods for recovering the attack surface of the complex system, executed in the virtual machines. Our contribution is the following: new method for recovering the runtime architecture of the system, new approach for finding executables in the virtual machine memory, and new hybrid introspection method for analyzing the execution of interacting compiled and interpreted programs. We implemented these new approaches for QEMU emulator running Linux-based OSes with CPython interpreters, as they widely used for creating complex multi-component systems.

Keywords–emulation, attack surface, dynamic analysis, software certification, taint analysis, virtual machine introspection

1. Introduction

Attack surface of the system consists of possible entry points, that can be used to breach into it. Attack surface of the single application usually divided into the following parts [3]:

- ways of communication with the program
- program code that controls the communication
- valuable data used in the application
- program code for protecting the valuable data

Detection of the attack surface is the significant part of the security development lifecycle (SDL), software testing, or software certification, because program code from the attack surface has higher requirements for reliability, security, and so on.

In this paper we focus on finding program code that can be treated as attack surface, due to processing of the input data. This code includes concrete functions, that get the data buffers directly, and the binary modules and applications, that could recieve the copy of the input through documented or unwanted channel.

A complex software information system usually includes many components: web server, web application with backend and frontend, database, firewall. Each of these components may be considered as a part of attack surface: network packets are processed by web server, then passed to web application, and finally written to database. Any of such software components (program modules) of the systems may be exploited by an attacker. For example, bug in Log4j library affected thousands of servers [8].

Small web application, which code can fit to one screen, deployed in the new environment, automatically grows up into a complex system with many interfaces [1], [12].

Attack surface could be made as small as possible to reduce the number of components that require attention [26]This approach is called "application debloating" and uses static and dynamic analysis to find the useless components.

But even when the system was "debloated", some of the useful modules can receive user-generated or sensitive data though direct function calls or side channels [5].

Existing approaches and tools focus only on separate components of information systems, but none of them examine system as a whole. Methods for finding the components, modules, and functions (i.e., the attack surface), that are activated in realistic scenarios, are not explored enough, and applied to the separate applications only. "Real world" full-system testing scenarios can reveal side channels of data transfer and allow finding more components of the attack surface, that can be debloated, analyzed, or isolated.

The real attack surface may include applications on scripting languages. Commonly used analysis methods can only map the executed binary code to the executables, using virtual machine introspection methods [9], [15], [18]. This code can be mapped to the source, using DWARF debug information, making the analysis convenient. But the script interpreters are different, they adds new runtime abstraction layer to the execution. The same binary code processes different "real" code parts from the script. Therefore it requires additional introspection layer

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00023

to make available the information about the script functions being executed.

As we are trying to make the attack surface as full as possible, we started with the following research questions:

- *Can we find which functions are executed in the VM?* Finding the functions is not as simple, as instrumenting the executed instructions, because mappings of the executables is not known, and there could be many versions of the same program in the system.
- *Can we introspect Python interpreter, executed within the VM?* We are targeting full-system analysis to allow reconstructing the attack surface for the virtual machines. But there are no existing approaches, that can extract the behavior of the interpreted Python program from the virtual machine.
- *Is full-system taint analysis useful for analysis of the Python programs?* We already used taint analysis to find the attack surface of the compiled programs [16]. Python applications can also be a part of the attack surface. But taint propagation works on the CPU instruction level, therefore we have to figure out how to fill this semantic gap.

The contribution, presented in the paper, is the following:

- Hybrid introspection method. This is a new introspection method for analysis of the systems that include interacting binary executables and scripts. It allows recovering the functions of the executed script code.
- Method for recovering the runtime architecture of the system. We used taint analysis and hybrid introspection to find the data dependencies between applications. It also allows finding the attack surface, which includes binaries and Python scripts.
- Method for detecting executed files in memory. It helps in finding the executed functions and distinct the similar executables, that have the same name, e.g., running in different containers.
- Case study of recovering the attack surface for simple web application, which processes the file, uploaded by the user. Our example demonstrate the usefullness of the presented approach — we can find the functions and modules that should be fuzzed (or debloated, or anything else), because they take part in processing the data, coming from the user.

2. Related Work

There are no public tools for finding the full stack system-wide attack surface. Even system architecture and data flows are usually recovered manually by the experts [13], [30].

2.1 Data Flow Analysis

Data flow tracking, or taint analysis, allows examining the flows of sensitive data in the system [21]. It may be used to detect data leakage, malicious code, and other unauthorized behavior.

Taint analysis was already used for taint tracking in the virtual machines [6], [9], [18] or even in the real machine, using the hardware co-processor [29]. Full system taint analysis allows tracking how data is processed inside the OS kernel, and tracking the data flow between the applications. Taint analysis itself may be used for different purposes, but these tools do not try using it for recovering the runtime architecture of the system. Such full-system taint analysis tools are targeted to detection of tainted code execution and analysis of the web applications.

2.2 Analysis of the Virtual Machines

There are debuggers that allows requesting information about the state of the virtual machine and events inside. One of such tools is `PyREBox` [22]. This QEMU-based debugger retrieves the list of running applications, modules, and functions. However, this information may be obtained only in the process of the interactive debugging and can't be used for automatic system architecture recovering.

`Cuckoo Sandbox` [7] is the open source automated malware analysis system. It is targeted to make different tracing jobs, like API or network tracing of a single application. Its capabilities for observing the runtime behavior of the complex systems are limited to file or network logging and memory dump analysis.

`Katana` is a tool for analysis of the virtual machine memory snapshots [15]. In contrast with `Panda` [9], `Rekall` [25], and other tools, it does not require a pre-defined OS profile to extract the information about the executed applications. However, it is targeted to forensics, when only one memory snapshot is available, and lacks the dynamic analysis and other applications, when benign software is under the attention.

2.3 Dealing with Attack Surface

Software debloating is the process of removing the unused features. Thus it decreases the program size and makes attack surface smaller [19], [23]. Papers that describe software debloating intended to be used with separate applications and can't be applied for complex systems that consist of many services that interact through files, pipes, and shared memory.

Attack surface should get an additional attention while testing the system. Dynamic testing of web application interfaces includes scanning the API, crawling with browser, and scanning the dependencies [12]. It can be used to identify unwanted interfaces, or verify the correctness of the API. But it can't look deeper and find the binaries, and their possible flaws, that process the data, coming through the API.

Another application of the attack surface is fuzzing the functions that were found. There are many tools for fuzzing the compiled applications [14], [27], [28] and Python scripts [2]. Fuzzers are rarely applied to the full application, due to the complexity of the control flow. Usually they test only one branch of the call graph, starting from the selected function. Therefore knowing the attack surface is the key to successful fuzzing.

3. Overall Natch Structure

Natch is a set of tools, that implement the approaches created during this research. It uses full-system dynamic analysis, because complex systems that consist of multiple applications.

that may be executed within containers, can't be analyzed with application-level instrumentation tools. We used QEMU full-system emulator for implementation of the attack surface analysis system. QEMU employs dynamic translation and allows creation of instrumentation layer upon the executed code.

Natch, our attack surface detection system, performs the following actions to recover the runtime system architecture and find the attack surface:

- Collect the information about executed processes.
- Monitor the network connections and file operations.
- Get addresses for loaded executable modules.
- Trace executed scripts.
- Analyze data flows from the tainted inputs.
- Visualize the analysis result and generate reports.

4. VM Introspection

Virtual machine introspection is used for obtaining information about the executables and runtime events inside the system [17]. There are several approaches to implementing introspection. We use non-intrusive approach to the introspection [11], which does not execute instrumentation code within the guest system, because it is compatible with VM execution replay.

To reconstruct the attack surface we need to recover the following information:

- Processes and their command line arguments. Program itself may be a part of an attack surface. Knowing which programs are executed is the first step for system debloating or thoroughly testing.
- Executables and their memory mappings. When full-system emulator executes the code, it deals with machine instructions. There is no information about the executable modules. To find the attack surface we have to recover executables and map executed instructions to the named functions in the binaries.
- File information. Some of the executables are not in the scope of analysis, but user may need to know their paths. File access logging is useful too, because this could be the part of the attack surface.
- Socket information. Network connection is usually the part of the attack surface. Knowing all information about open sockets helps in improving the application security.

Introspection approach described in [11] is not powerful enough for collecting all the information about processes, tasks, files, and memory mappings. Therefore we extended it by creating the parser of the internal kernel data structures. Similar approach with parsing the kernel data structures was already implemented in [9]. This is achieved by collecting some addresses of the kernel data structures and offsets of their fields. The set of such values is called "kernel profile", because it depends only on kernel build: source code version and compilation parameters.

We introspect the kernel data structures to build the application-level view on the virtual machine. The following Linux data structures were tracked:

- `task_struct` fields to get information about tasks and processes, their relations, command lines, and runtime states.
- `files_struct` fields for recovering the file properties when we inspect file-related system calls.
- `cred` structure fields to get the process user ids and capabilities.
- `mm_struct` and `vm_area_struct` are used for building process memory map and finding the executable names loaded.
- `socket` structure fields for retrieving the assigned addresses and other data from open sockets.

All these structures are monitored during VM state change events, that can be coupled with change of the application parameters. E.g., when CPU returns from kernel mode to user mode, it means that the previously active process could exit or can be destroyed, therefore we should check the `task_struct->state` field.

Our implementation of the introspection do not require running guest instrumentation code, making it compatible with deterministic execution replay. Deterministic replay allows one to record system execution and then to reproduce it over and over as many times as it is needed [10]. As the analysis may slowdown the emulator for the several times, the deterministic replay is especially important to mitigate the impact of this slowdown on the guest system behavior.

With all collected system information we can build runtime state of the process tree, list of open files, and so on. But even when we know the name of the executables, we can't be sure, that these are the executables of our interest. Therefore we need a method for finding the executables' mappings.

4.1 Finding Executables in the VM Memory

Virtual machine execution consists of continuous switching between the kernel code and application code. Therefore user can observe (through logging or debugging) the flow of the machine instructions and can't assign those instructions to the processes or executable files.

Using VM introspection from the previous section, we can distinguish the executed instructions between the processes. But in the most cases each process includes many executables: program itself and shared libraries. Allocation of the executable memory areas can't be determined statically, because the loader may assign any addresses for the binaries (and this is the preferred mode, ASLR, needed to mitigate the attacks). We used kernel file structures and description of the memory areas to distinguish the executed modules. But this approach gives only file names, assigned to memory areas. This does not help in finding the functions that was actually called, because executable section offset within the memory area may be unknown. And logging the functions in runtime is required to collect the attack surface reports. This method also does not help in detecting the loaded kernel objects, because process memory mappings do not include kernel information.

Mapping of the obtained executable path to the binary (and its symbols) is not easy to implement for the following reasons:

- Different containers within the virtual machine can include different builds of the same program.
- Guest disk image should be unpacked to verify the paths. This is an additional time-consuming operation, which does not completely solves the problem, due to the following.
- Files may be extracted or copied at runtime, therefore their paths will not exist in the unpacked image.
- Correct code sections offsets still have to be determined somehow, due to ASLR. Section offsets are needed for finding the functions, that were executed in the VM.

That is why we invented a new method, which determines the mapping of the analyzed binaries to the guest memory. The executables and the VM memory have the common unchanged part: executable code. We compare bytes, that can be extracted from the mapped pages in the VM, with the bytes extracted from the analyzed files. Let E be the number of executables, S — their size in bytes, and P — size of a single code page. Our mapping method should comply the following requirements and limitations:

- Time complexity of the code page processing should be $O(P)$ in average. There should be no additional delay related to large number of executables or their code section size. This requirement can only be satisfied with state machine or hash table.
- Method should be capable of detecting large 4 Gb executables and and small 16 kb kernel objects.
- Sample executables can be pre-processed, but their set could be changed (e.g. adding one new executable) without any significant re-hashing efforts.

The fastest possible search can be implemented with the state machine (or trie) which uses the contents of the executable code pages as input bytes. Every byte moves trie pointer (or state of the machine) forward, until getting the prefix, which matches only one executable. However, this method has several drawbacks:

- Code pages are not static. Some of the bytes are assigned by the loader, and their values depend on executable base address. These bytes are called "relocations".
- State machine for the analyzed set of the executables can't be cached, because new executables may be added (or some may be removed). And rebuilding the state machine would be $O(S)$.
- Fastest machine with shortest prefixes will make many false positive errors. E.g., if there is a code with first byte `0x42` in a single executable, all pages starting with this byte will be recognized as belonging to this file. Even when there are other guest files, that were not used for matching.

The first drawback does not allow treating the code sections as a long sequence of bytes. If we just skip the relocations, we get many chunks of code bytes, but every chunk has its own offset. Therefore executable matching complexity may reach $O(P \cdot E)$.

To decrease the complexity, we decided to split the executable code sections into samples of the same length L and offsets aligned to L. All samples with relocations inside were just dropped.

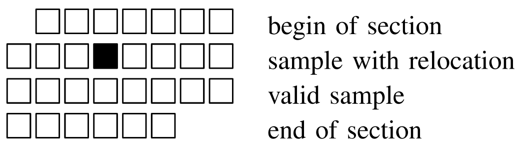


Figure 1: `.text` page loaded into code page and divided into several samples.

With this simplification, we can calculate hash values for all samples in $O(S)$ time during preparation stage. On execution phase, we just load all hashes into the hash table in $O(S/L)$ time (which is much better than $O(S)$ recalculation).

But how big (or small) should be the sample size L? To figure it out, we took the `/usr` directory in Ubuntu 20.04 and scanned all executables inside. There were 9476 executables and dynamic libraries. For $L = 64$ there were 48966228 samples with unique hash (88% of total). For $L = 128$ there were 24476712 samples with unique hash (94% of total). For $L = 256$ there were 12224197 samples with unique hash (96% of total).

Collision level should be as low as possible, but all these levels are normal, because these are collisions between the samples. But every code page usually has several valid samples. And if first one is not unique, the second one will probably identify the executable.

We wanted sample size to be as small as possible to allow working with dynamically loaded kernel objects, that have tiny code sections. Therefore we have chosen $L = 128$, which has low enough collision level. Usually the number of analyzed executables is below 100, therefore there are no collisions at all.

Our new executable module detection method can be used for commodity executables, for large files (like Linux kernel), and for small files (like kernel objects). It was successfully applied to locate the executed functions in the running VM.

4.2 Debug Info and Executed functions

When all executables are mapped, we can inspect their code as it is executed. Attack surface, that we recover, should include the list of the functions, that processed the input data, and the context of their invocations, i.e. backtrace.

One of the approaches for creating the backtrace is instrumenting the `call` and `ret` instructions within the virtual machine. But in some cases it is not enough. Consider the example from [24] (Figure 2).

Executable loader puts the addresses of the dynamic library functions into global offsets table (GOT). When the program needs to invoke such functions, `call` instruction proceeds to the entry in `.plt` section. That entry loads actual function address from GOT and jumps to that address.

`Call` instruction goes to `puts@plt` function in `test` executable, therefore we'll have this function in the call graph instead of desired reference to `libc.so`. To solve this problem, we read debug information for loaded libraries, and

```
0334 <.plt>:
0334: ff b3 04 00 00 00 pushl 0x4(%ebx)
033a: ff a3 08 00 00 00 jmp *0x8(%ebx)
...
0354 <puts@plt>:
; indirect jump to address from GOT
0354: ff a3 10 00 00 00 jmp *0x10 (%ebx)
035a: 68 08 00 00 00    push $0x8
035f: e9 do ff ff ff    jmp 334 <.plt>

043c <test>:
...
; load GOT address into ebx
0448: 81 c3 ac 1b 00 00 add $0x1bac, %ebx
...
; call to trampoline in .plt
0465: e8 ea fe ff ff    call 354 <puts@plt>

1ff4 <.got.plt>:
2004: 00 00 5a 03
```

Figure 2: Trampolines in `.plt` section for calling the dynamic library functions.

instrument entry points of all functions. When control flow reaches one of these addresses, we switch the call graph entry to the new name.

This method also allows dealing with tail function calls, when `call` instruction is replaced with `jmp` by the compiler. Such optimization does not allow detecting the correct (source-level) call graph, but our instrumentation of the function entry points at least allows us not to miss the executed function at all (Figure 3). Therefore our instrumentation allows creating the attack surface, which includes the correct function names in all similar cases.

5. Hybrid Introspection

When we explore the attack surface of the application, we can find which functions process used-supplied data and recover the call chain for them to find out the components that affected by the called functions. But when the application includes interpreted code, we can only see some interpreter-related functions (Figure 4), but not the information about the user code, which is written on JavaScript, Python, or PHP [4], [20].

When such kind of system is analyzed, the user expects to get list of the script functions, not the functions of the interpreter.

We started with Python, because it is widely used as a language for creating the web application backends. Python virtual machine interprets the script code, saving references to the source files, because they can be used for debugging.

To recover the information about the executed Python functions and program modules, we created new non-intrusive method for introspection of the Python interpreter, running inside the virtual machine.

First, we extract debug information from compiled Python interpreter (or download this information when interpreter is taken from system packages). This information includes function names and addresses, type and variable description.

With function names we can reconstruct the stack as on Figure 4. Some of these functions process the source code function calls. We found in CPython code all the functions that are used to process script function calls. In most cases these functions take `PyObject` pointer as a parameter. This is a base object type and it includes a pointer to `PyTypeObject`, which describes the real type of the object.

We analyzed all embedded types and built an introspection code to extract names and source references from any of them. Using this information, we can produce backtraces, that include the executed functions of the Python scripts (Figure 5).

With our new hybrid introspection, we can either annotate the calls of the C functions with Python names, or build separate call graph with Python functions only. This method also helps in determining what dynamic libraries are invoked within the script, or how interpreted code processes the input data, received from other services running within the system.

6. Taint Analysis for Attack Surface Detection

Attack surface includes the entities that can process user's data. VM introspection can identify the processes that work in the system, executables that were loaded, resources that were acquired by that processes. New hybrid introspection method can also look into scripts and trace the interpreted functions. But without any data flow tracking we can't find out the executables and functions that were actually used in data processing (and not just run in parallel).

Therefore identifying the attack surface for the system includes the following steps:

- Taint sensitive source data.
- Propagate the taints as the data have been processed.
- With VM introspections identify the processes, that receive tainted data.
- With mapped executables detection, find out the functions, that process tainted data.
- Find call chains that lead to tainted data processing.
- With hybrid introspection recover the list of Python scripts and functions, that also get the process the tainted data.

6.1 Taint Tracking

Prior applications of taint analysis to virtual machines were targeted to detecting buffer overflow-like vulnerabilites [18]. When vulnerability is found, that code is definitely the part of an attack surface. There also could be other parts, that can be attacked, but exploits for them are not found yet. Finding such parts for thorough testing with fuzzing is the aim for our taint analysis applicaton [16].

Finding the attack surface is getting the list of the functions, dynamic libraries, and executables, that process the input data. In Natch sensitive data can be read from selected TCP/UDP port, or specific file within the VM. The concrete source can be selected through the configuration file.

Target application do not access tainted buffer directly. And values from that buffer may be passed to other applications,

```
int function1(int d)
{
    int x = d * 10;
    return function2(x);
}
```

(a) C code

```
function1:
  lea   edi, [rdi+rdi*4]
  add   edi, edi
  call  function2
  ret
```

(b) Non-optimized code

```
function1:
  lea   edi, [rdi+rdi*4]
  add   edi, edi
  jmp   function2
```

(c) Optimized code

```
Caller
    function1
        function2
```

(d) Source-level view at call graph

```
Caller
    function1
```

(e) Call graph with `call` instrumentation

```
Caller
    function1
    function2
```

(f) Call graph with entry point instrumentation

Figure 3: Tail call optimization adds an obstacle for building call graph. Entry point instrumentation allows not to miss executed functions, while finding the attack surface.

```
_PyFunction_Vectorcall (Objects/call.c:334)
  _PyEval_EvalFrameDefault (Python/ceval.c:1578)
    _PyObject_MakeTpCall (Objects/call.c:174)
      slot_tp_call (Objects/typeobject.c:7482)
        _PyObject_Call_Prepend (Objects/call.c:407)
          _PyObject_FastCallDictTstate (Objects/call.c:121)
            _PyEval_EvalFrameDefault (Python/ceval.c:1578)
              _PyObject_MakeTpCall (Objects/call.c:174)
                slot_tp_call (Objects/typeobject.c:7482)
                  _PyObject_Call_Prepend (Objects/call.c:407)
                    _PyObject_FastCallDictTstate (Objects/call.c:121)
                      _PyEval_EvalFrameDefault (Python/ceval.c:1578)
                        _PyFunction_Vectorcall (Objects/call.c:334)
```

Figure 4: Backtrace example for the Python interpreter.

```
MiddlewareMixin.__call__ (django/utils/deprecation.py:90)
  SessionMiddleware.process_request (django/contrib/sessions/middleware.py:18)
    cached_property.__get__ (django/utils/functional.py:72)
      WSGIRequest.COOKIES (django/core/handlers/wsgi.py:116)
        get_str_from_wsgi (django/core/handlers/wsgi.py:207)
          get_bytes_from_wsgi (django/core/handlers/wsgi.py:194)
            encode
          decode
        parse_cookie (django/http/cookie.py:10)
          split
          strip
```

Figure 5: Call graph example for the Python functions. Functions embedded into the Python interpreter do not have reference to the source code.

which should be also considered as a part of an attack surface. Therefore we should propagate the taints as the memory cells are copied. Full system taint analysis does not need to identify the interfaces used to pass the data, because all these interfaces are implemented as some sequence of CPU instructions, that process the data.

6.2 Process and Function Identification

With taint propagation we can find the instructions of any of the processes, that access the tainted data. To make it useful for attack surface detection, we taint only copy operations destination, when source value is tainted, and not taint the result of arithmetic and logic operations. The rationale for this behavior is the following: if the attacker wants to construct an exploit, this exploit will probably consist of some unchanged buffer from the input. And when the input is modified, the exploit is not so easy to construct.

Therefore, highest priority for testing and debloating, should receive the code, that gets the unchanged values from the input. VM introspection allows us to identify currently executed process, by reading current `task_struct` pointer from the kernel. When virtual CPU reads or writes tainted data, we can check the current process, and include it into the attack surface.

But process name, extracted from `task_struct` is ambiguous, many different applications or versions may have the same name. That is why we implemented the executable detection subsystem. It uses the new method, which can handle large and small executables. Executable modules, extracted from the kernel, or detected with our new method, are the first subject for testing or debloating. But we also want to inspect the system behavior deeper.

After detecting the executable mappings, we can find out the executed functions. With the mappings of the executed functions, we can include them into the attack surface, when they access the tainted data. But information about direct taint accessors is not always useful enough.Tainted data in many cases are accessed with `memcpy` or similar functions, that called from higher-level wrappers.

That is why the our main report for attack surface analysis is call graph. With that graph, one can inspect the function call chain, which leads to tainted data usage, and select the functions to be tested with fuzzing.

6.3 Tainted Python Functions Identification

The compiled function is identified as tainted, when it accesses the tainted data. But scripted functions do not make memory accesses directly. These functions consist of interpreted operators, that executed with the help of the interpreter's functions. This is similar to the case, illustrated by Figure **??** in the previous section. As the Python function execution needs an access to the tainted data, which is performed by some helpers, this function should be considered tainted.

We highlight such functions in the call graph, therefore user can choose them or select one of their callers for the fuzzing. Let's look at `hmac_new` function at Figure 6. This is an embedded function in module `_hashlib`. This function is located in `_hashopenssl.c` CPython source file. This function constructs new object and calculates the hash of the input key, using openssl library.

```
HMAC.__init__ (hmac.py:38)
  HMAC._init_hmac (hmac.py:66)
    hmac_new
```

Figure 6: Backtrace fragment for `hmac_new` function call.

We consider `hmac_new` as tainted, because from the Python point of view, it reads the value of the key argument. This is done indirectly, but such tainting decision is consistent with function's header (Figure 7).

```
_hashlib.hmac_new
    key: Py_buffer
    msg as msg_obj:
        object(c_default="NULL") = b''
    digestmod:
        object(c_default="NULL") = None
Return a new hmac object.
```

Figure 7: `hmac_new` description from the CPython standard library.

Therefore hybrid introspection allows us generating the attach surface report, which includes processes, modules, functions, and Python functions, that process the sensitive data.

7. Runtime Architecture Recovering

Using the information about the tainted data propagation and the executed processes, modules, functions, Natch builds several kinds of reports.

The first one, is the report with process tree, which is similar to the output of `ps --forest -ax` (Figure 8). It includes all processes, that were activated within the analyzed run, and their parents.

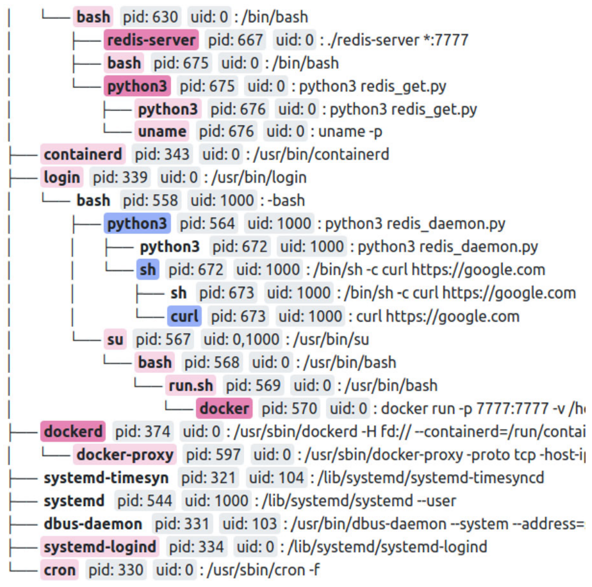
The second one, which reveals the overall sensitive data processing stack, is the report with data flow between the processes (Figure 9). Data flows are shown with arrows. Processes, sockets, and files presented as nodes of that graph. Red-colored processes have root privileges.

Third report discovers the sensitive data flow between all executable modules, including kernel, dynamic libraries, applications (Figure 10). It also shows files and network sockets as tainted data input and output. These recovered modules should be the first target for fuzzing and debloating, because they interfere (directly or not) with user's data.

The last two reports show call graph for the functions in binary code (Figure 4), and for Python functions (Figure 5). Both of these reports cover only the call chains, that ended with sensitive data processing.

All the reports listed above allow us to observe several levels of the attack surface:

- All processes that were executed.



```
|    └── bash  pid: 630  uid: 0 : /bin/bash
|        ├── redis-server  pid: 667  uid: 0 : ./redis-server *:7777
|        ├── bash  pid: 675  uid: 0 : /bin/bash
|        └── python3  pid: 675  uid: 0 : python3 redis_get.py
|            ├── python3  pid: 676  uid: 0 : python3 redis_get.py
|            └── uname  pid: 676  uid: 0 : uname -p
├── containerd  pid: 343  uid: 0 : /usr/bin/containerd
├── login  pid: 339  uid: 0 : /usr/bin/login
|    └── bash  pid: 558  uid: 1000 : -bash
|        ├── python3  pid: 564  uid: 1000 : python3 redis_daemon.py
|        |   ├── python3  pid: 672  uid: 1000 : python3 redis_daemon.py
|        |   └── sh  pid: 672  uid: 1000 : /bin/sh -c curl https://google.com
|        |       ├── sh  pid: 673  uid: 1000 : /bin/sh -c curl https://google.com
|        |       └── curl  pid: 673  uid: 1000 : curl https://google.com
|        └── su  pid: 567  uid: 0,1000 : /usr/bin/su
|            └── bash  pid: 568  uid: 0 : /usr/bin/bash
|                └── run.sh  pid: 569  uid: 0 : /usr/bin/bash
|                    └── docker  pid: 570  uid: 0 : docker run -p 7777:7777 -v /h
├── dockerd  pid: 374  uid: 0 : /usr/sbin/dockerd -H fd:// --containerd=/run/contai
|    └── docker-proxy  pid: 597  uid: 0 : /usr/sbin/docker-proxy -proto tcp -host-i
├── systemd-timesyn  pid: 321  uid: 104 : /lib/systemd/systemd-timesyncd
├── systemd  pid: 544  uid: 1000 : /lib/systemd/systemd --user
├── dbus-daemon  pid: 331  uid: 103 : /usr/bin/dbus-daemon --system --address=
├── systemd-logind  pid: 334  uid: 0 : /lib/systemd/systemd-logind
└── cron  pid: 330  uid: 0 : /usr/sbin/cron -f
```

Figure 8: Fragment of the process tree. It includes privileged processes (red) and processes that accessed tainted data (blue and dark red).

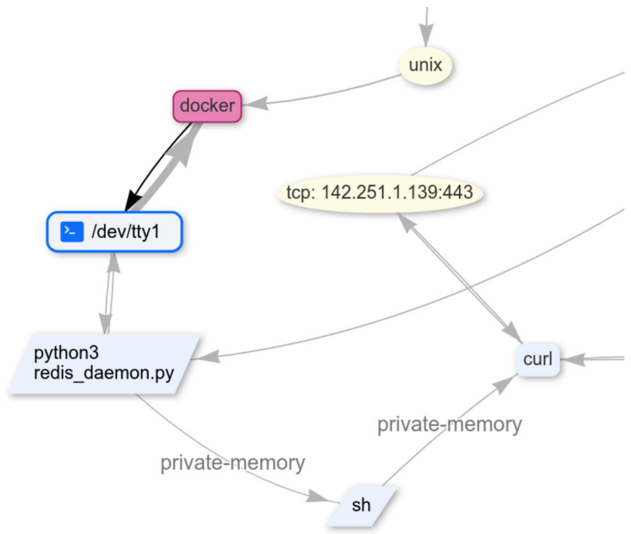


Figure 9: Tainted data flow through processes, files, and sockets.

- Processes, that work with sensitive data.
- Dynamic libraries and executables, that work with sensitive data.
- Functions in the executables, that accessed the sensitive data, and their call chains.
- Python functions and scripts, that worked with sensitive data, and their call chains.

8. Case Study

We wanted to choose some real life (or similar) application with many components, but not too heavy to analyze. Therefore we have chosen a web application, based on Django.

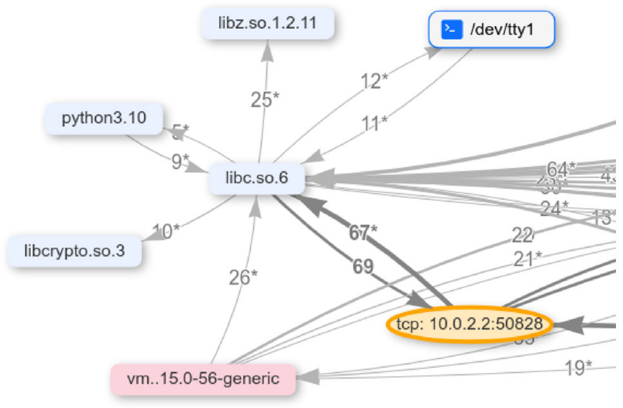


Figure 10: Tainted data flow through dynamic libraries and executables, data files, and sockets.

Django is the complex Python-based framework for creating web applications[1]. Its marketing description includes the phrase "you can take web applications from concept to launch in a matter of hours". The opposite side of this kind of simplicity, is the huge amount of code inside the framework itself and packages, required by Django.

If one needs to reuse some Django-based web component in the sensitive application, then this component should be thoroughly tested. But which code to test? Attack surface detection methods should give us the answer.

We used sample Django-based application[2] to demonstrate the usefulness of system-wide data flow analysis for analysis of the combination the interconnected Python scripts and binary programs.

We set up the virtual machine with Ubuntu 22.04 OS and installed the web sample into it. User is intended to login into that program through the web interface and use the database connected to the application backend. One of the possible actions for the user — importing the CSV files, that include list of the records, into the database. Every record includes string, integer, and date fields.

The contents of the uploaded CSV file was tainted and we examined the resulting reports, generated by Natch.

First, we discovered the architecture of CSV processing workflow: packet comes from the external network and processed by Linux kernel, libc socket functions, Python interpreter, and SQLite database (Figure 11).

This figure shows, that some portions of the tainted data are processes by SQLite. We can examine the call graph and find the SQLite functions, responsible for that: `_pysqlite_query_execute` from CPython dynamic library, `sqlite3VdbeExec` from `libsqlite3.so`. These functions may be fuzzed as a part of attack surface, because they receive user-generated data in the parameters. Finding bugs is not the aim of our research, therefore we will not dive

[1]https://www.djangoproject.com/

[2]https://github.com/app-generator/sample-django-charts-argon

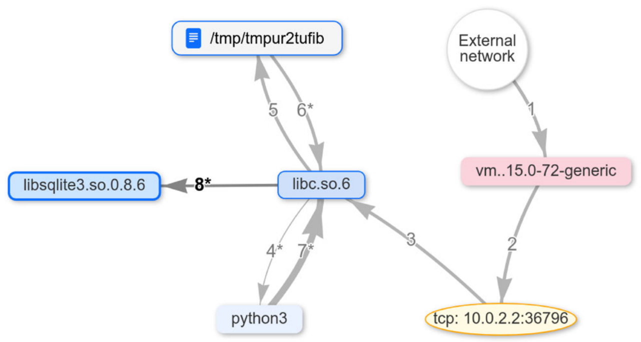


Figure 11: Executable modules that process the input CSV data. Red one is the Linux kernel. There is also a temporary file, which is used in data processing.

too deep into the fuzzing.

Python interperter itself is not in the analysis scope, because it uses script code and input network traffic (including target CSV) as an input. Script code changes the logic of the analysis, because different kinds of input data may pass through the same interpreter functions. That is why we reconstructed call graph for the Python functions and found the following call stack entries that used the tainted data directly or though the called helpers:

- `_strptime` — parsing the date and time from the input (CPython standard library).
- `CSV.create_dataset` — converts CSV stream into `Dataset` object (djando-import-export package).
- `SQLInsertCompiler.prepare_value` — prepares a value to be used in SQL query (SQL compiler from Django).
- `SQLiteCursorWrapper.execute` — converts inputs to be passes to SQLite database (SQLite adapter from Django).
- `Template.compile_nodelist` — compiles HTML page with uploaded data (template compiler from Django).

After that, one may want to analyze or test these functions. E.g., first function, `_strptime`, may be fuzzed with the following code[3]:

```
#!/usr/bin/python3
from _strptime import _strptime
from pythonfuzz.main import PythonFuzz

@PythonFuzz
def fuzz(buf):
    try:
        string = buf.decode("ascii")
        _strptime(string)
    except UnicodeDecodeError:
        pass
    except ValueError:
        pass
```

[3]We haven't found any bugs in `_strptime`. This is a function from standard CPython library, and is probably well-tested already.

```
if __name__ == '__main__':
    fuzz()
```

The similar fuzzing wrappers may also be created for other functions and classes, with some efforts for creating the correct context.

We also recovered the complete list of Python application modules and dependencies, that can be used as a target for debloating. They were collected as for the whole "login-upload-logout" scenario, and only for part, which covers "upload CSV-get reply".

The complete exported report and some screenshots from interactive analysis pages can be observed at https://github.com/Dovgalyuk/QRS2023_materials.

9. Conclusion and Future Work

In this work we created some new methods for virtual machine introspection and attack surface analysis. These methods were implemented within the Natch tool. This tool allows recovering the runtime architecture and finding the attack surface for the complex systems, that include many interconnected binary executables and scripted programs, written on Python. Natch was tested with most of the popular Linux distributives and all commodity CPython versions.

After the successful implementation of the hybrid introspection for the CPython, we plan to extend it to other scripting languages, like JavaScript and Lua, or bytecode/JIT machines like Java and .NET/Mono.

References

[1] A Seismic Shift in Application Security, 2020 https://about.gitlab.com/resources/downloads/gitlab-seismic-shift-in-application-security-whitepaper.pdf

[2] Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris

[3] Attack Surface Analysis Cheat Sheet, https://cheatsheetseries.owasp.org/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.html

[4] Bonfante, G., Ithayakumar, A. Where is the Virtual Machine Within CPYTHON?. In: Jourdan, GV., Mounier, L., Adams, C., Sèdes, F., Garcia-Alfaro, J. (eds) Foundations and Practice of Security. FPS 2022. Lecture Notes in Computer Science, vol 13877. Springer, Cham.

[5] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. 2019. Principled Unearthing of TCP Side Channel Vulnerabilities. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/3319535.3354250

[6] Chow, Jim and Pfaff, Ben and Garfinkel, Tal and Christopher, Kevin and Rosenblum, Mendel. Understanding Data Lifetime via Whole System Simulation. Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04, 2004, San Diego, CA, 22–22, 1, USENIX Association, Berkeley, CA, USA

[7] Cuckoo Foundation. Automated Malware Analysis, https://cuckoosandbox.org/

[8] CVE-2021-44832, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832

[9] B. Dolan-Gavitt, T. Leek, J. Hodosh, W. Lee. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 2013.

[10] P. Dovgalyuk, 2012 16th European Conference on Software Maintenance and Reengineering, Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging, 2012, 553-556

[11] Dovgalyuk, Pavel and Fursova, Natalia and Vasiliev, Ivan and Makarov, Vladimir, QEMU-based Framework for Non-intrusive Virtual Machine Instrumentation and Introspection, Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, 2017, Paderborn, Germany, 944–948, 5, ACM, New York, NY, USA

[12] Dynamic Application Security Testing (DAST), https://docs.gitlab.com/ee/user/application_security/dast/

[13] S. Fayyad and J. Noll, "A framework for measurability of security," 2017 8th International Conference on Information and Communication Systems (ICICS), Irbid, Jordan, 2017, pp. 302-309, doi: 10.1109/IACS.2017.7921989.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: combining incremental steps of fuzzing research. 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020

[15] Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus. ACM, New York, NY, USA, 18 pages.

[16] N. Fursova, P. Dovgalyuk, I. Vasiliev, M. Klimushenkova and D. Egorov Detecting Attack Surface With Full-System Taint Analysis, 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Hainan, China, 2021, pp. 1161-1162, doi: 10.1109/QRS-C55045.2021.00174.

[17] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Proc. Network and Distributed Systems Security Symposium, 2003, 191–206

[18] Henderson, Andrew and Prakash, Aravind and Yan, Lok Kwong and Hu, Xunchao and Wang, Xujiewen and Zhou, Rundong and Yin, Heng, Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform, Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA, 248–258, 11, ACM, New York, NY, USA,

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS), 2018.

[20] Obi Ike-Nwosu. Inside The Python Virtual Machine, 2020

[21] Lok Kwong Yan and Andrew Henderson and Xunchao Hu and Heng Yin and Stephen McCamant. On Soundness and Precision of Dynamic Taint Analysis. Syracuse University, 2014, SYR-EECS-2014-04

[22] Python scriptable Reverse Engineering Sandbox, a Virtual Machine instrumentation and inspection framework based on QEMU, https://pyrebox.readthedocs.io/en/latest/

[23] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In Proceedings of the 28th USENIX Security Symposium, 2019

[24] Redirecting Functions in Shared ELF Libraries, https://www.apriorit.com/dev-blog/181-elf-hook, 2014

[25] Rekall memory forensic framework, http://www.rekall-forensic.com

[26] Maryam Rostamipoor, Seyedhamed Ghavamnia, Michalis Polychronakis. Confine: Fine-grained system call filtering for container attack surface reduction. Computers & Security, Volume 132, 2023

[27] K. Serebryany. Continuous fuzzing with libFuzzer and AddressSanitizer. 2016 IEEE Cybersecurity Development (SecDev), page 157. IEEE, 2016

[28] A. Vishnyakov, D. Kuts, V. Logunova, D. Parygina, E. Kobrin, G. Savidov, A. Fedotov. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle. 2022 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE

[29] Yu Wang, Jinting Wu, Haodong Zheng, Zhenyu Ning, Boyuan He, Fengwei Zhang. Raft: Hardware-assisted Dynamic Information Flow Tracking for Runtime Protection on RISC-V. 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'23), Hong Kong, October, 2023.

[30] George O. M. Yee. 2019. Modeling and reducing the attack surface in software systems. In Proceedings of the 11th International Workshop on Modelling in Software Engineerings (MiSE '19). IEEE Press, 55–62.