

Identification of API Functions in Binaries

 synthesis.to/2023/08/02/api_functions.html

Twitter Training About Me

02 Aug 2023 - Tim Blazytko

During my presentation “Unveiling Secrets in Binaries using Code Detection Strategies” at REcon 2023 ([slides](#), [recording](#), [code](#)), I showcased heuristics how to navigate in unknown binaries in various reverse engineering settings, such as malware analysis, vulnerability discovery and embedded firmware analysis. In this talk, I also presented a simple but powerful technique to identify common API functions in statically-linked executables and in embedded firmware. This blog post will delve deeper into this subject, exploring these API functions, the intuition behind the heuristic, and its additional use cases in the context of malware analysis.

In reverse engineering, API calls provide crucial insights into a program’s behavior. They often perform high-level operations like file access, network communication, string operations or memory management. They help us reverse engineers to build a better understanding of what a program does and how it works. In the context of vulnerability research, we can locate areas of code that are commonly associated with vulnerabilities, like functions related to memory management or user input processing. For malware analysis, we can learn how a malware sample interacts with the operating system, networks, files, etc.; this can help us to understand its purpose and functionality.

However, in some cases, API functions are hard to identify in binaries: for example in statically-linked executables and in embedded firmware. In statically-linked binaries, the code of standard libraries and third-party libraries is incorporated directly into the binary. Since there is no clear separation between the user code and the libraries, without symbols, it is difficult to identify which functions belong to the program and which are from libraries. In the case embedded reverse engineering, firmware is often designed to directly interact with the hardware of a specific device; therefore, it may contain unique API functions which contain similar functionalities as in standard libraries, but are not present in those. Sometimes, manufacturers even use their own proprietary standard library implementations; without access to the source code or documentation, it can be nearly impossible to identify these functions.

In this blog post, we will initially discuss common methods for detecting API functions; then, we’ll evaluate the heuristic and showcase additional use cases for malware analysis. So, if you are curious how to detect API functions in various reversing scenarios, stay along. If you want to play around with the heuristic on your own, you can use my [Binary Ninja plugin obfuscation_detection](#).

Methods to Detect API Functions

Without symbol information, it remains challenging to identify API functions in statically-linked binaries and embedded firmware for the reasons mentioned above. In practice, two common approaches rely on signature matching and cross-referencing with known libraries.

The concept behind *signature matching* is straightforward: it involves searching for specific byte sequences—or signatures—of known API functions within a binary. These signatures, stored in a database, each act as a distinct fingerprint for an associated API function. During analysis, the binary is scanned and the byte sequences of functions are compared to the signatures from the database. When a match occurs, it signifies the presence of that particular API function in the binary. Several systems employ this function signature method, such as [IDA Pro's Fast Library Identification and Recognition Technology \(FLIRT\)](#) and Binary Ninja's [Signature Kit Plugin](#).

However, the efficacy of signature matching heavily depends on the accuracy of the signature database. This database must include signatures corresponding to the specific library version used in the binary; furthermore, it must be compiled for an identical platform (operating system & CPU architecture) using similar compiler settings. Variations can cause significant differences and lead to mismatches, limiting the usability of such systems in practice.

The second method, *cross-referencing with known libraries*, employs a comparable strategy. In this method, binary diffing techniques (e.g., [BinDiff](#)) are used to compare the binary against different versions of known libraries to locate similarities. When a match is found, the function name and type information from the known library can be transferred to the binary under investigation. Since this method considers additional features like control-flow graph structures and call hierarchies, it allows slight differences in library versions and compiler settings without causing a complete mismatch; this way, it potentially overcomes some limitations of the signature matching approach.

As a downside, this method often carries a significant degree of uncertainty due to the reliance on feature matching, which can lead to false positives and negatives. Furthermore, cross-referencing is still constrained by the availability of a corresponding library. If the binary was compiled with an uncommon or proprietary library for which there is no available version to compare against, these methods would not be effective.

In the following, we will take a look at another simple but effective heuristic, which has proven to be very efficient in many reversing scenarios.

Heuristic: Frequently Called Functions

One heuristic approach with notable potential is the analysis of call frequency: identifying functions within a binary that are invoked frequently by others. This simple technique can yield substantial insights: functions with high call frequencies are often of significant interest in the reverse engineering process, since they often represent core functionalities that the software leans heavily upon.

Often, these frequently called functions are crucial API functions, facilitating a wide range of fundamental operations within a program. Examples include string operations (e.g., `strlen`, `strcmp`), memory management functions (e.g., `malloc`, `free`, `memcpy`, and `memset`), file access methods (e.g., `open`, `close`), and network communication (e.g., `send`, `receive`). These API functions play pivotal roles within the binary, making them frequent call targets and providing valuable insights into the binary's operation. Therefore, identifying these high-frequency call targets can offer valuable footholds for understanding the inner workings of the binary.

To implement this heuristic in your disassembler of choice, intuitively, we sort all functions by their number of independent callers in descending order. To focus on the most relevant findings, one might choose to concentrate, for example, on the top 10% of these functions.

In BinaryNinja, this operation can be achieved within a few lines of Python code:

```
def find_most_frequently_called_functions(bv):
    print("Frequently Called Functions")

    # print top 10% (iterate in descending order)
    for f, num_callers in get_top_10_functions(bv.functions, lambda f:
len(f.callers)):
        print(
            f"Function {hex(f.start)} ({f.name}) is called from {num_callers}
different functions.")
```

In the code above, the function `find_most_frequently_called_functions` takes a binary view `bv` (a representation of the analyzed binary) as input, and applies the `get_top_10_functions` function to the list of all functions within the binary. This function ranks the API functions based on the number of independent callers; the result is then printed for the top 10% of functions. This approach surfaces those functions that are most frequently called, making them prime candidates for further investigation during the reverse engineering process.

Similar implementations can be realized in IDA or Ghidra: In IDA, the corresponding functions can be found by iterating over cross-references to a particular function; in [Ghidra](#), the `getCallingFunctions` method of a function object can be utilized to identify the callers.

However, while our outlined heuristic offers an intuitive approach for identifying API functions in binaries, let us now investigate whether we can validate its effectiveness in various real-world reverse engineering scenarios.

Evaluation

To evaluate the effectiveness of our heuristic in real-world scenarios, we will apply it to a set of diverse case studies:

- the `ls` command from the Linux [Coreutils](#) package,
- a statically-linked malware sample from the [XOR DDoS family](#),
- an embedded firmware and
- another malware sample from the [PlugX family](#).

For consistency and comparability, we will limit our analysis to the top 10 identified functions in each case. We anticipate that these functions will reveal commonly invoked API calls. Moreover, our heuristic may also uncover critical functions that underpin the key functionalities or logic of the binary under analysis. In these cases, we'll delve deeper to understand their roles and operations. With this said, let's plunge into the results!

Coreutils

Our first case for evaluation is the `ls` command from the Coreutils package, well-known for generating command line directory listings. Since this binary is dynamically linked, API functions appear as wrappers that enable jumps to functions not included in the binary; at load time, the exact function locations are resolved by the operating system during the dynamic linking phase.

Upon applying our heuristic, we obtain the following results:

```
Function 0x100007528 (_putchar) is called from 28 different functions.
Function 0x100007522 (_printf) is called from 23 different functions.
Function 0x100007558 (_snprintf) is called from 12 different functions.
Function 0x1000074bc (_getenv) is called from 10 different functions.
Function 0x10000755e (_strcmp) is called from 9 different functions.
Function 0x1000069f0 (sub_1000069f0) is called from 9 different functions.
Function 0x100007582 (_strlen) is called from 8 different functions.
Function 0x100007564 (_strcoll) is called from 8 different functions.
Function 0x10000745c (_err) is called from 8 different functions.
Function 0x1000073fc (___error) is called from 8 different functions.
```

A quick scan at the results reveals that 9 out of the top 10 identified functions relate directly to API calls. These include operations related to string manipulation (such as `_strcmp`, `_strlen`, `_strcoll`, `_snprintf`), output to `stdout` (`_putchar`, `_printf`), environment variable parsing (`_getenv`), and error handling (`_err`, `___error`).

Interestingly, one function in the top-10 list, `sub_1000069f0`, does not appear to be a typical API function. However, a closer examination reveals that this function acts as a wrapper for the API function `_putchar`:

```
1000069f0 55          push    rbp {__saved_rbp}
1000069f1 4889e5     mov     rbp, rsp {__saved_rbp}
1000069f4 e82f0b0000 call   _putchar
1000069f9 31c0      xor     eax, eax {0x0}
1000069fb 5d        pop     rbp {__saved_rbp}
1000069fc c3        retn   {__return_addr}
```

In summary, all of the top 10 identified functions are API-related.

XOR DDos

Our second case study involves a statically-linked [malware sample](#) from the XOR DDos family. As indicated by its name, this Linux-targeting malware executes [DDos attacks](#) and employs XOR-based encryptions (for strings and C&C server communication). This particular sample retains all its symbols; therefore, it serves as an excellent evaluation target, as we can use these function names to easily confirm whether the identified functions are API functions. The output of our heuristic test on this sample is as follows:

```
Function 0x8065320 (free) is called from 298 different functions.
Function 0x8066a70 (memcpy) is called from 191 different functions.
Function 0x80662b0 (strlen) is called from 184 different functions.
Function 0x80669b0 (memset) is called from 174 different functions.
Function 0x8063d30 (__libc_malloc) is called from 151 different functions.
Function 0x8053810 (__lll_unlock_wake_private) is called from 148 different
functions.
Function 0x8053700 (__lll_lock_wait_private) is called from 122 different functions.
Function 0x8060080 (ptmalloc_init) is called from 114 different functions.
Function 0x80569b0 (__strtol_internal) is called from 99 different functions.
Function 0x80661f0 (strcmp) is called from 93 different functions.
```

The most frequently called function is `free`, an API function which deallocates memory on the heap. Other frequently invoked API functions are also related to memory management (`__libc_malloc`), data movement (`memcpy`), memory initialization (`memset`) and string operations (`strlen`, `__strtol_internal`, `strcmp`). Moreover, the appearance of API functions like `__lll_unlock_wake_private`, `__lll_unlock_wake_private` and `ptmalloc_init` among the top-called functions suggests that the malware extensively employs multi-threading. This behavior aligns well with the nature of the sample, which establishes a multitude of network connections to execute DDos attacks. Once again, all identified functions are API functions.

Embedded Firmware

For our next case study, we delve into embedded firmware reverse engineering. We examine a non-further specified demo application designed for an embedded microcontroller operating on an [ARMv8M architecture](#). This application implements hardware initialization routines as well as basic network communication logic relying on cryptographic techniques. Instead of using a standard library, the firmware employs a custom statically linked library. We've compiled the firmware with the highest level of optimization, while retaining function symbols for easier analysis.

These are the results:

```
Function 0x8081cd8 (memcpy) is called from 379 different functions.
Function 0x808140c (vLoggingPrintfError) is called from 343 different functions.
Function 0x8081d26 (memset) is called from 338 different functions.
Function 0x80625a4 (sys_assert) is called from 307 different functions.
Function 0x80785ae (DbgConsole_Printf) is called from 250 different functions.
Function 0x8040738 (multiply_casper) is called from 247 different functions.
Function 0x807e2a0 (ulSetInterruptMask) is called from 160 different functions.
Function 0x807fd28 (CASPER_MEMCPY) is called from 139 different functions.
Function 0x80821e0 (__mbedtls_mpi_free_veneer) is called from 102 different
functions.
Function 0x80511b8 (mbedtls_platform_zeroize) is called from 87 different functions.
```

The heuristic reveals a wild mix of the most frequently called functions within the embedded firmware: At the top of the list, we find `memcpy` and `memset`, reflecting an extensive amount of memory manipulation operations throughout the firmware. This is followed by `vLoggingPrintfError` and `DbgConsole_Printf`, which are associated with console output functions, implying a logging or debugging system implemented within the firmware, while `sys_assert` hints towards a custom error-handling mechanism.

The firmware further includes custom routines to handle memory-related tasks, such as `CASPER_MEMCPY`, `__mbedtls_mpi_free_veneer` and `mbedtls_platform_zeroize`. These functions belong to the cryptographic libraries [CASPER](#) and [mbedTLS](#), providing optimized or secure alternatives to standard memory operations. Notably, `mbedtls_platform_zeroize` offers a secure means to erase sensitive data from memory, a crucial function in cryptographic contexts.

Additionally, `multiply_casper` provides an optimized version for cryptographic multiplications. Furthermore, `ulSetInterruptMask` appears to be related to the firmware's interrupt handling system, a common element in microcontroller-based applications.

In summary, our heuristic provides insights into the firmware's core functionality by highlighting a mixture of standard library functions, application-specific error handling routines, console output functions, custom cryptographic implementations, and functions related to interrupt management within the firmware.

PlugX

For our final experiment, we take a look at a [sample](#) from the PlugX malware family. Compared to the other case studies, due to the lack of function symbols, we have significantly less guidance. To put it differently, we solely rely on our heuristic and manual analysis.

Here are the results obtained from the PlugX sample:

```
Function 0x10001620 (sub_10001620) is called from 1253 different functions.
Function 0x10001590 (sub_10001590) is called from 1253 different functions.
Function 0x1002cc5e (__seterrormode) is called from 320 different functions.
Function 0x1002cc4e (sub_1002cc4e) is called from 219 different functions.
Function 0x1002d530 (__free_base) is called from 141 different functions.
Function 0x100302f0 (sub_100302f0) is called from 90 different functions.
Function 0x10001530 (sub_10001530) is called from 73 different functions.
Function 0x10003af0 (sub_10003af0) is called from 63 different functions.
Function 0x10004360 (sub_10004360) is called from 61 different functions.
Function 0x10003cb0 (sub_10003cb0) is called from 31 different functions.
```

Given these results, two observations stand out: Firstly, the only two identifiable API functions are `__seterrormode` and `__free_base`; these are responsible for error handling and memory management. Secondly, two functions—`sub_10001620` and `sub_10001590`—are called with a significantly high frequency compared to all other functions in the binary.

Analyzing the calling patterns of these functions reveals a fascinating interplay: Whenever `sub_10001620` is invoked, `sub_10001590` has always been executed just a few lines before. Even more, the input parameters of `sub_10001590` are string values corresponding to Windows APIs, such as `kernel32`. Finally, we also note that the return value of this function is fed as input to `sub_10001620`, indicating that the output of the one function serves as input to the other; if we put everything together, we can represent a typical instance of this call structure as follows:

```
sub_10001620(0, sub_10001590("kernel32"), 0xffc97c1f)
```

We now understand why both functions are invoked an equal number of times: They are interdependent. Yet, upon closer inspection, we make another noteworthy discovery. The third parameter of the outer function—the constant `0xffc97c1f`—in combination with the string `kernel32`, suggests that this sequence of function calls represents an *API hashing routine*, a mechanism often utilized by malware to obfuscate API calls from static analysis. This mechanism works by traversing a series of API function names represented as strings, computing their hash values and subsequently comparing the computed hash values to a pre-computed constant during runtime. If a match is found, the corresponding API function is imported and executed.

Without going too much into detail, further analysis reveals that `sub_10001590` acts as an obfuscated call to `LoadLibraryA`, which loads a specified Windows library into the address space of the calling process. Subsequently, `sub_10001620` implements a slightly altered version of the `CRC32` algorithm, which compares the computed CRC32 hash to the constant input (`0xffc97c1f`). This workflow perfectly matches the modus operandi of API function hashing.

Using this knowledge, we can now represent the function calls as follows:

```
crc32(0, LoadLibraryA("kernel32"), 0xffc97c1f) ; "GetProcAddress\x00"
```

In this context, the constant `0xffc97c1f` corresponds to the hash of the `GetProcAddress` function, which is loaded from `kernel32`.

To conclude this case study, our heuristic once again identified all key API functions within the analyzed malware sample. Although it doesn't pinpoint specific API functions as explicitly as it does in the other case studies, the heuristic confirms the underlying assumption that API functions are called frequently. In this particular case, an API hashing routine—designed to import hidden API functions—emerged as the most frequently called function structure in the entire malware sample. As a consequence, the heuristic pinpoints code locations which provide a deep understanding of the malware's behavior; therefore, it can serve as a potent tool in malware analysis.

In a broader perspective, we can say the heuristic is very effective in detecting frequently used API functions. Even in instances where highlighted functions are no API calls, they often expose other integral functionalities of the binary—for example, cryptographic operations in the embedded firmware sample. The heuristic proves to be not just simplistic and easy to implement, but also extremely efficient. It provides deep insights into binaries, irrespective of the reverse engineering scenario at hand, be it malware analysis, embedded firmware reverse engineering or the analysis of statically-linked executables; it is an invaluable tool in our reverse engineering arsenal.

Closing Remarks

In a series of [previous blog posts](#), I've discussed various efficient, architecture-agnostic heuristics that allow us to pinpoint obfuscated code in large binaries. Throughout these blog posts, I've often highlighted that these heuristics have a wider scope, proving valuable across a diverse range of reverse engineering scenarios. These observations inspired me to present at REcon, showcasing the versatility of these heuristics in a plethora of contexts. This blog post marks another significant step on that path.

Interestingly, the heuristic introduced today represents a clear departure from previous ones, as it supports a wide range of reverse engineering scenarios, while it is the first not specifically designed to pinpoint obfuscated code. With the same goal of providing useful

tools and insights for the entire reverse engineering community, I've updated the documentation of my [obfuscation detection plugin](#) to offer more intuitive use cases and insights for everyday reverse engineering tasks. If you're interested, I encourage you to check it out.

My professional work does not only span the domain of code (de)obfuscation—a topic I deeply explore in my [training sessions](#)—but also traverse the landscapes of embedded security and malware analysis. Therefore, you can expect the introduction of more potent heuristics designed for a broad spectrum of reverse engineering scenarios over time. So, stay tuned for more exciting things to come. Until our next encounter, thanks for all the fish! :)

Contact

For questions, feel free to reach out via Twitter [@mr_phrazer](#), mail tim@blazytko.to or various other channels.