TaintBlade: A framework for automated protocol reverse engineering and study of malware command and control protocols

Marcos Sánchez Bajo
University Carlos III of Madrid
Madrid, Spain
100405823@alumnos.uc3m.es

Juan Manuel Estévez Tapiador
University Carlos III of Madrid
Madrid, Spain
jestevez@inf.uc3m.es

***Abstract*—The exchange of information between applications heavily relies on communication protocols, which define the rules and conventions for message exchange. However, many software applications rely on closed, proprietary protocols that lack publicly disclosed specifications. This is particularly true for malware, where developers intentionally hide protocol details to impede reverse engineering by analysts.**

In this work, we present TaintBlade, a powerful protocol reverse engineering framework that utilizes dynamic execution tracing techniques to infer the protocol message format of arbitrary programs. It features (1) an instrumentation module that seamlessly hooks into loaded images, executed routines, and unique instructions, providing access to registers and memory data; (2) a tainting module with a multi-color scheme that precisely tracks data propagation at the byte level within the instrumented program; (3) a heuristics module that matches executed instructions with tainted data to a set of heuristics corresponding to specific operations; (4) a protocol reversing module that constructs a full protocol based on the gathered heuristics; (5) a tracing module that detects the execution of routines and logs their arguments; and (6) a noper module that allows for skipping code sections and to manually modify the program state at arbitrary points.

TaintBlade aims to be a usable and practical solution for the analysis of command and control (C2) protocols in malware. It also features a GUI that provides a user-friendly interface for seamlessly navigating the data produced by the tool.

***Index Terms*—Protocol reverse engineering; Command and control; Taint analysis; Intel Pin; Malware analysis**

I. Introduction

In today's interconnected world, software applications rarely work in isolation. The need for communication has become prevalent as programs often depend on exchanging data and messages with other remote entities. This sparked the creation of network protocols, that serve to define the rules, format and other conventions that follow the messages exchanged in the communication.

Although many protocols are open and documented, it is common to find software using closed, proprietary protocols whose specifications are undisclosed. This is particularly true in the case of malware, where developers intentionally keep the protocol details hidden to hinder reversing efforts undertaken by malware analysts.

Nowadays, a significant number of malware samples incorporate a command and control (C2) protocol that dictates the communication between the malware and the C2 server. Therefore, the need for reverse engineering these protocols has become increasingly relevant in the field.

Protocol reverse engineering is the process of studying a communication protocol with the ultimate goal of uncovering its underlying structure and inferring its implementation, including the format of its messages and the grammar of the communication. Classically, this process is mainly done manually, which is both inefficient and time-consuming. Even for malware running simple protocols, we can find numerous technical challenges during the analysis, like obfuscation, that hinders easy access to the protocol details. Additionally, protocols frequently undergo modifications across different versions of the same malware family, further increasing the time to be spent by analysts. Therefore, there exists a great interest in developing techniques for automating protocol reverse engineering that are able to programmatically infer the underlying protocol of an arbitrary program without access to the source code.

The existing techniques for performing automated protocol reverse engineering are varied and can be grouped into two big categories depending on the techniques employed: those relying on execution traces and those using network traces [1] [2].

- Execution tracing refers to the instrumentation of a program using dynamic analysis techniques with the goal of capturing internal information about the process of the binary executable. In the case of malware, this usually requires setting up a sandboxed environment and the simulation of the C2 server, that might be unavailable at the time of analysis.
- Network tracing involves analyzing the communication using traffic capturing tools, such as Wireshark or tcpdump. These tools dump the raw bytes sent and received during the communication, which can then be analyzed with static analysis techniques.

Independently on the techniques used, the ultimate goal of

protocol reverse engineering tools is to produce a model of the protocol, which is usually represented either by a Protocol Format (PF), a Protocol Finite State Machine (PFSM), or both.

- PFs represent the messages exchanged in the protocol, which are partitioned into fields through a process known as message format inference. A field refers to a group of bytes that share syntactic information, meaning that each field is delimited and classified depending on how the analyzed program interprets it during the execution [5] (e.g. a field could be a keyword that is checked during a C2 communication). When the message format inference is done using dynamic tracing techniques, it can also include semantic information regarding the implementation of the binary (e.g. which routines are in charge of interpreting the message), which enriches the syntactic meaning of each field (e.g. the field is not only a keyword, but also a command that the program executes). We explain message format inference in more detail in section II-A.
- PFSMs represent the exchange of messages in the protocol as a state machine. Following a process of protocol grammar inference, representing the protocol as a PFSM allows for including information about the time and order of messages, displaying the rules that govern the exchange of messages in the protocol.

The decision to choose between a PF and a PFSM depends, as we discussed, on the nature of the data we aim to represent. However, there are advantages and disadvantages associated with selecting either execution tracing or network tracing techniques.

Techniques based on network tracing are usually time-sensitive, easy to operate and do not require to instrument the binary, which is particularly advantageous when dealing with malware. However, they only have access to the transmitted messages, lacking information about the internal program functionality. This means that, unlike execution tracing techniques, they cannot be used for inferring the semantics of the message fields [3].

On the other hand, execution tracing is usually slow, requires the execution of the binary, and needs of accurate instrumentation of the program, which is dependent of the architecture of the binary. Despite these challenges, execution tracing techniques provide access to internal program information, enabling the inference of message field semantics.

Project goals. As we have identified, the development of protocol reverse engineering tools, specifically for the analysis of command and control protocols in malware, has gained significant attention. Consequently, a significant amount of research has emerged focusing on various methods for inferring protocols' messages and grammar. However, we observed that many of these projects are either proof of concepts or remain with their code unavailable. Recent surveys on the matter [3] [4] reveal that, in the recent years, only some projects such as Nemesys [11], Prisma [9], Pulsar [10], Netzob [12] and Reverx [8] are open and accessible. Some projects, like Polyglot [5], have released partial source code for certain components. And among these projects, only Netzob is actively developed. As a major drawback, it focuses on static analysis techniques, therefore lacking some of the semantic information only accessible using dynamic analysis, as we previously identified.

In this work, we present our automated protocol reverse engineering framework, called *TaintBlade*, that leverages execution tracing techniques to infer the protocol message format of arbitrary programs. The development of this project pursues three main goals: (1) to accurately infer the underlying ingress protocol message format of a program, incorporating semantic information derived from dynamic analysis to represent its fields and their purpose in the protocol; (2) to provide a comprehensive trace of program activities, including loaded images, child processes, executed routines, and arguments, assisting analysts in the reversing process rather than offering a black-box tool; (3) to serve as an open-source malware analysis framework, offering specific functionalities for studying malicious behaviors, whilst catering to the general public with an intuitive and feature-rich graphical user interface that enhances cohesion over the generated data.

This initial version of *TaintBlade* is available for Windows systems and is prepared to work with the x86 and x86_64 architectures.

II. Background

This section introduces all the background needed for our research into automated protocol reverse engineering and introduces the reader to the underlying techniques implemented in our tool.

A. Message format inference

In this section, we will introduce basic concepts about the message format inference process that we have implemented in *TaintBlade*. It must be noted that we are presenting standard terminology that is also common to other works like ReverX [8] or Polyglot [5]. We will go into detail on how these fields are detected during the description of our framework.

Message. A message is the data unit over which message format inference is applied. It consists of a group of bytes that are either sent or received in a single exchange of information in the communication. For example, in an HTTP botnet, if a C2 server sends a command via a GET request, then all the bytes in the request are grouped in a single message.

Field. The result of the protocol message format inference process is the partition of each message into fields. A field is a consecutive sequence of bytes that form a distinct unit within the message's syntax. When semantic information is accessible, then a field also shares semantic information, implying that all bytes within the field serve a common purpose during the program execution. In the previous example of an HTTP botnet, a field could represent the command being transmitted.

Inferring a field depends on its type. Next, we will cover the types of fields and the common approach towards their detection in a message.

Delimiter. A delimiter is a constant field used as a separator, marking the boundaries between other fields, particularly when these are of variable length. A delimiter can consist of one or more bytes, and its value is predetermined by the protocol. For instance, in the JSON format, it is common to find the symbol ":" for separating a key and its corresponding value.

Detecting a delimiter through dynamic analysis techniques involves examining the specific bytes that the program analyzes in the message at runtime. Since the program is unaware of the exact location of the delimiter within the message, it typically compares the delimiter value with each byte of the message. These comparisons are performed sequentially, always using the same delimiter value.

Keyword. A keyword is a constant field that is fixed by the protocol. Although it may or may not appear in the transmitted message, its value is known a priori by the program and always checked at runtime. For example, in an HTTP GET request, the request is indicated by the starting keyword "GET".

Detecting a keyword using dynamic analysis involves analyzing the program's executed comparisons during runtime. Unlike delimiters, these comparisons are performed only once rather than repeatedly or sequentially. Given that long keywords may exceed the processor word size, multiple assembly-level comparisons may be employed to check a single keyword. Consequently, reversing keywords requires taking into account the compared data, since multiple comparisons against consecutive bytes may need to be considered as a group.

Pointer fields. A pointer field is a field, usually numeric, that indicates the position of another field in the message. This is common for indicating the length of a variable-length field, whose length is unknown to the program a priori.

During the program execution, we will find that the value of a pointer field is usually added or subtracted to a pointer variable, specifying an offset into the message. Therefore, using dynamic analysis, we may detect a pointer field whenever we find that the value of a byte from the message is used to specify an offset into another byte of the message.

Variable-length field. A variable-length field is a field that is designed to accommodate an arbitrary length. Usually we can find its boundaries defined by a delimiter, or its length indicated by a pointer field. When available, we can assign a common semantic meaning to each byte within the field by examining the routines utilizing the field.

We can attempt to find variable-length fields by looking at the positions in the message pointed by pointer fields.

Fixed-length field. A fixed-length field is a field with a constant length. Its starting position may be specified by a delimiter, or it may just appear sequentially after another keyword or field. Similar to variable-length fields, we can assign a common semantic meaning to each byte within the field by examining the routines that utilize the field.

A field may be classified as fixed-length following a process of elimination (when it does not correspond to any other field class) or when we know that bytes with a specific semantic meaning have a fixed length. For instance, if we detect that a field is being used as an IP address, then we know that the field has a constant length.

B. Dynamic Binary Instrumentation

Our protocol reverse engineering framework *TaintBlade* is built on top of Intel Pin [19], a dynamic binary instrumentation (DBI) tool developed by Intel.

DBI is a powerful technique consisting on the instrumentation and modification of the instructions of a program during its execution. This is achieved using dynamic code injection techniques (in the case of Intel Pin, a DLL is injected in the instrumented process), allowing for the analysis of the runtime behavior of the program. This includes observing the executed instructions and routines, accessing and modifying the contents of registers and memory, and the insertion at runtime of new instructions to execute.

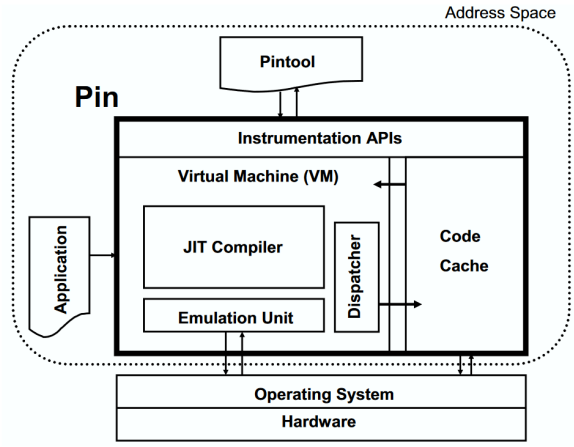

Fig. 1: Intel Pin architecture, from [20].

Figure 1 shows an overview of Intel Pin's architecture. At any time, there are three programs running: the application that will be instrumented, Pin itself, and a Pintool. We will now explain how the instrumentation process works so that we understand these components and how they interact between each other.

Once Pin's DLL is injected in the application to instrument, it intercepts the first instruction of the program. From that point on, Pin uses a Just-In-Time (JIT) compiler to process the assembly code of the program, one basic block at a time. This component compiles the received code and stores it in a code cache. Then, using a dispatcher, Pin transfers the execution flow to the generated sequence. When the basic block ends, Pin proceeds to take the next one from the application. Additionally, Pin utilizes an emulator to execute certain special instructions that cannot be directly executed, such as specific system calls.

Pintools are the instrumentation component of Pin. These are C/C++ programs (developed by the user using Pin) that make use of Pin's instrumentation API. A Pintool is commonly made of two types of instrumentation routines:

- Instrumentation routines, that are executed every time the JIT Compiler fetches code from the application. They allow for accessing the static components (like the instructions being executed), but not the dynamic ones (like the value of the registers when the instruction executes).
- Analysis routines, that are injected from instrumentation routines. These are called whenever Pin executes, from the code cache, a specific code point. They have access to the dynamic components, such as the value of the registers after the instruction execution.

TaintBlade's main component is a Pintool and, therefore, it makes use of a combination of instrumentation and analysis routines.

C. Taint analysis

Our framework *TaintBlade* features a novel tainting engine. In this section, we will briefly explain the basics of the taint analysis technique.

Taint analysis is a program analysis technique that allows for determining the influence that a selected program state has on other parts of the program state [17]. In the context of a C2 communication, this would allow us to determine whether the data received from the network (the initial code state, this data is said to be 'tainted') is used in any way at the time of executing a command (the final code state).

Taint analysis usually involves three steps: defining taint sources, defining taint sinks, and the specification of some taint propagation rules.

- Taint sources are program locations where some data is tainted. These are usually routines, whose arguments or return value are tainted.
- Taint sinks are program locations where we check whether they are influenced by any previously tainted data.
- Taint propagation rules determine how the taint propagates from the taint source through the program execution. Every executed instruction is instrumented, and evaluated to check whether it must spread the taints somehow (or untaint some data). For example, if a register *rax* is tainted, and we execute the instruction *add rbx, rax*, then now register *rbx* is tainted too.

Taint analysis is usually slow, and requires an efficient management of the taint data, which as the program is executed it can get exponentially large. Traditionally, taint storage has been done using shadow memory and bitmaps, where a value of 0 indicates that a register/memory is not tainted and a value of 1 indicates that it is tainted. This is called mono-color tainting. As we will explain, *TaintBlade* features multi-color tainting which allows for distinguishing the different sources from which taint data is originated.

III. Design

In the previous sections, we introduced the issue of automating protocol reverse engineering and the relevance of this task in the context of the analysis of malware transmitting malicious communications. Additionally, we discussed some techniques, such as message format inference and taint analysis, that we will be using in our tool.

In this section, we will now describe the automated protocol reverse engineering framework that we have developed, called *TaintBlade*.

A. Architecture overview

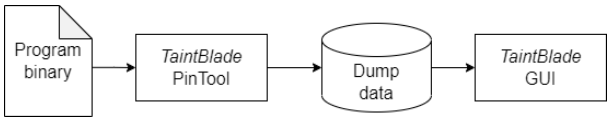

Fig. 2: General architecture of TaintBlade

From a general viewpoint, the functioning of *TaintBlade* is, as shown in Figure 2: (1) the user selects a binary (an executable) to be traced; (2) the program is executed and instrumented by *Intel Pin*, using *TaintBlade* as a Pintool, which reverses the protocol of the program; (3) the Pintool generates a set of output *.dfx* dump files and/or a SQLite database with all traced data; (4) the user can navigate the resulting data by means of *TaintBlade* GUI, which interacts with the database, or by accessing the output dump files.

The *TaintBlade* Pintool is the central component of the framework. It encompasses all the necessary instrumentation and tracing functionalities required for the protocol reverse engineering task. At its core, the Pintool utilizes Intel Pin, which provides the instrumentation capabilities. On top of Intel Pin, *TaintBlade* features a series of *modules*: collections of components centered on a specific stage of the protocol reverse engineering task or that offer other additional functionalities intended to facilitate malware reversing.

As it can be observed in Figure 3, *TaintBlade* comprises six different modules, namely (1) an instrumentation module that seamlessly hooks into loaded images, executed routines, and unique instructions, providing access to registers and memory data; (2) a tainting module with a multi-color scheme that precisely tracks data propagation at the byte level within the instrumented program; (3) a heuristics module that matches executed instructions with tainted data to a set of heuristics corresponding to specific operations; (4) a protocol reversing module that constructs a full protocol based on the gathered heuristics; (5) a tracing module that detects the execution of routines and logs their arguments; and (6) a noper module that allows for skipping code sections and to manually modify the program state at arbitrary points

Although each module works independently of the others, most of them operate in a pipeline fashion, where the input of one module depends on that of the one directly preceding it. In particular, the instrumentation, tainting, heuristics and protocol reversing modules work sequentially towards the protocol reversing task, whilst the tracing and noper modules work separately, offering support features to the tool. We will now study each module individually and describe how they work internally and the coordination between them.

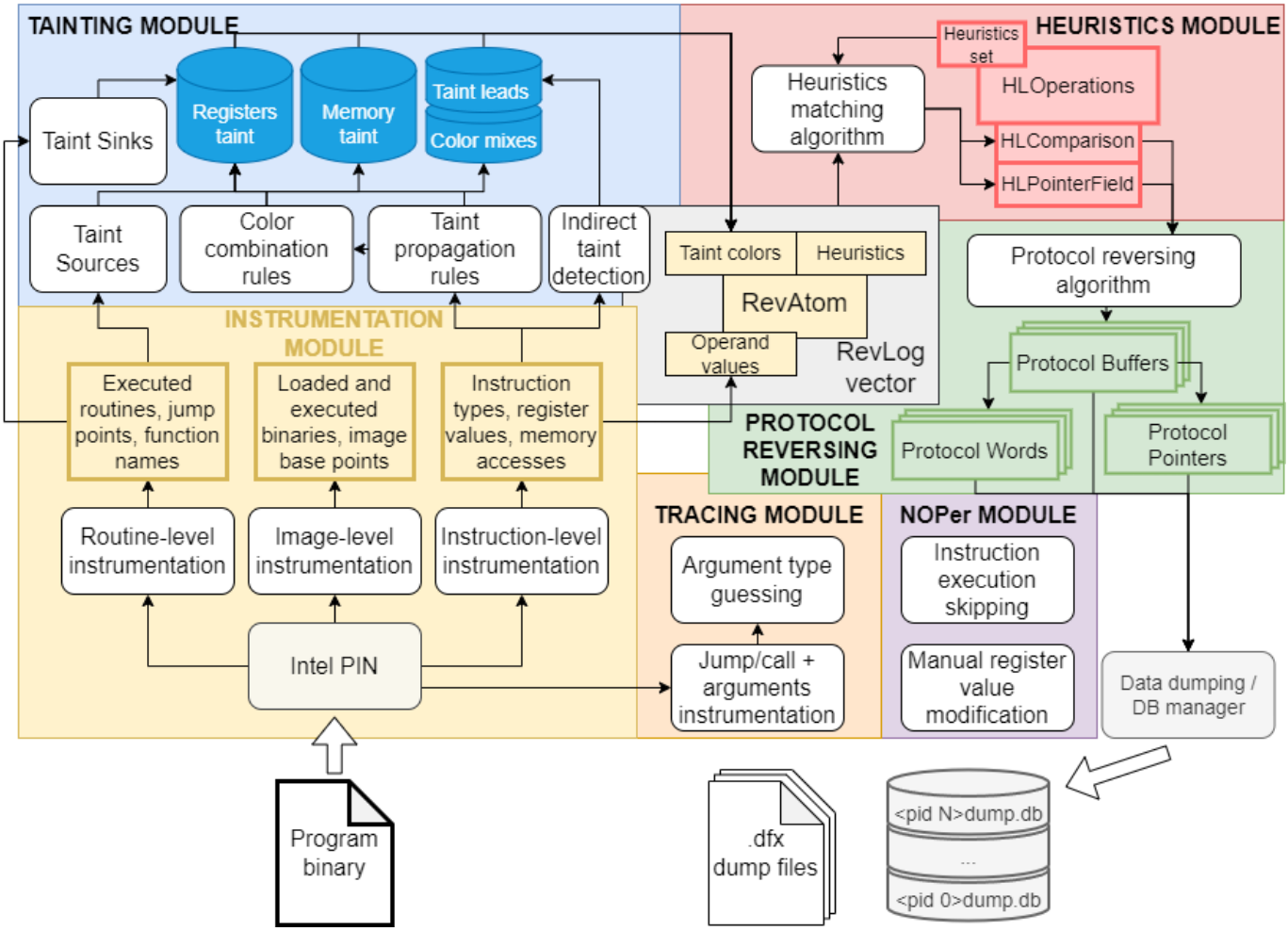

Fig. 3: Architecture of *TaintBlade*.

B. Instrumentation module

The instrumentation module is the central component in the *TaintBlade* framework and the one that works closer to the assembly code. It utilizes Intel Pin to instrument the program that is being executed at a triple level: creating hooks for every loaded program image (e.g. an imported DLL), for every executed routine (e.g. the *main* function at the program) and for every executed instruction (e.g. a single call instruction to some other function).

Image instrumentation. *TaintBlade* only instruments the instructions and routines that are included in a list known as the *image scope*. This scope is, by default, the main image of the executable selected to be traced, although the user may select more images to be included via the GUI or the program configuration files. By instrumenting program images, *TaintBlade* informs the user of when a new image is loaded - which may be of interest to the user, choosing it to be included in the scope. This instrumentation also acts as support to other modules -like the noper module-, by calculating in advance the virtual addresses of key instructions (such as those selected by the user to be manually skipped).

Routine instrumentation. *TaintBlade* instruments any newly executed routine included in the image scope. The instrumentation is double: it instruments the routine right before it is called and after it returns. This enables the tool to extract information such as the function and image names, and the memory address of the arguments with which each function is called and with which it returns. Instrumenting routines is key for supporting the rest of the modules, and it is done in two different ways:

- Comparing the name and image of the routine with an internal list of routines. The instrumentation of these routines is hardcoded, and therefore the tool knows exactly how to parse each of its arguments (the data structures each one corresponds to). This type of instrumentation is used from the tainting module, which needs to access specific data points in each routine at a specific time (either before or after the execution).
- Performing a more general instrumentation, without hard-coding the data types. As we will explain in later sections, this is useful for the tracing module, which will also attempt to guess the type of arguments later.

Instruction instrumentation. This is the most important instrumentation type since it is the entrypoint for the tainting functionality. Here *TaintBlade* will examine each executed instruction individually, and compare it with a list of instructions that the tool is prepared to work with. Table I covers the instruction types currently supported by *TaintBlade*. It must be noted that, for this version of the tool, we have focused on supporting those instructions that we considered to be more commonly generated by compilers (and thus more commonly found at any program). Certain instructions (like SHR, REPNE SCAS), although seemingly arbitrary, often take part at the implementation of common routines found at most programs, like *strncmp()* and *strlen()*, which is the reason they were selected to be instrumented.

TABLE I: Instruction types for which instrumentation is supported

Instruction type	Instruction	Description
Arithmetic operations	ADD SUB	Any addition of memory, immediate or register operands
Logical operations	AND OR XOR	Any logical operation between memory, immediate or register operands
Shift operations	SHR	Shifting X number of bits from a register to the right. Supports shifts of multiples of 8, since tainting module runs byte-based tracking.
Comparison operations	CMP	Comparison between memory, immediate or register operands
Data moving operations	MOV MOVSX MOVSXD MOVZX	Move values between memory, immediate or register operands.
Address moving operations	LEA	Moving a pointer, all types of LEA operands supported
Control flow instructions	CALL JMP JB JBE RET	Any jump, call or return. Supported both near and far relative displacement types
String manipulation operations	SCASB SCASD SCASQ SCASW REPNE SCASX	Instructions used for operating with string types, supports iterations with REPNE prefix

When an instruction is instrumented, *TaintBlade* first checks the type of instruction by making use of the Intel XED decoder [21] that comes embedded in Intel Pin. This decoder lets the tool classify the instruction, grouping it with other instruction types that share the same properties (e.g. every arithmetic operation is instrumented similarly). Once classified, the tool analyzes the arguments of the instruction and extracts any needed addresses and/or values from it. This data will be then used as an input for the tainting functionality.

C. Tainting module

TaintBlade features a custom multi-color tainting engine for the x86 and x86_64 architectures, operating at the byte-level. The rationale for this is that we needed to track and distinguish each byte received from a specific code program point. By employing multiple colors, we can then gather key information regarding how these bytes are combined between each other and how they are used inside the program.

This initial version of the tool is fully prepared to work in Windows systems, although it could be easily extended to work in Linux in a future. It must also be noted that we decided to develop the tainting system from the ground (instead of using an existing solution) because we did not find any implementation of a taint system for x86_64 Windows machines that included multi-color taint tracking. We considered well-known engines like libdtf [14], Triton [16] or Dytan [7] but, as indicated in Table II, we identified certain drawbacks in each of them.

TABLE II: Existing open-source tainting engines, not selected

Engine	Characteristics	Drawbacks
libdft	Linux x86 Multi-color (max. 8 colors)	No Windows, x86_64 support Limited number of colors
libdft64 (Angora)	Linux x86_64 Mono-color	No Windows support Multi-color not supported
Dytan	Linux x86 Mono-color	No windows, x86_64 support Multi-color not supported
Triton	Windows, Linux x86, x86_64, ARM32, AArch64 Mono-color	Multi-color not supported

We would like to highlight the difficulty of modifying these existing systems to support our purposes. Adapting an engine to a new architecture requires creating new taint rules for the new Instruction Set Architecture (ISA). Moreover, moving from a 32-bit-sized architecture to a 64-bit-sized one involves additional challenges. One of them is that tainting engines in the x86 architecture usually feature (like in libdft) a shadow memory with the same size as that of the whole virtual address space (limited to 2^{32} bytes in x86), which tracks the taint state of each byte. This is completely unfeasible to do in a 64-bit system, since the virtual address space can be up to 2^{64} (although processor implementations may reduce this value [18]).

Another challenge is the implementation of multi-color taint tracking in a mono-color system. The introduction of one of such systems involves changing the taint storage from a bitmap value, where 0=not tainted and 1=tainted, to a data structure that supports multiple values, together with all taint propagation policies in order to support this change. Additionally, we would find that the number of possible colors is limited to the capacity of the structure previously used.

Therefore, taking all the previous into account, we decided to create our own completely new 64-bit-compatible multi-color tainting engine that works with Intel Pin (it relies on the instrumentation module we described previously).

Taint storage. The tainting module keeps a centralized set of data structures that store the taint information of the registers and memory of a single instrumented process. Since

Fig. 4: Taint storage data structures.

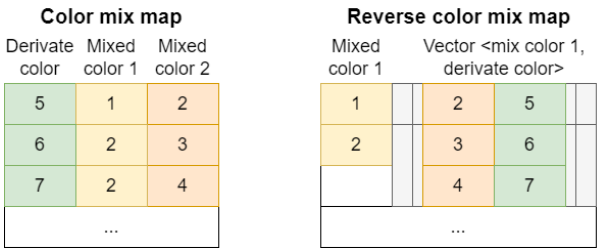

Fig. 5: Taint colors management data structures.

TaintBlade supports both the x86 and x86_64 architectures, it is not feasible to maintain a shadow memory of the process with the color of each byte. Instead, we achieved an optimized implementation that pursues two main goals:

- Since we cannot shadow all memory in the process, we dynamically generate or destroy taint data at runtime, thus giving up on temporal optimization but allowing to taint any number of memory addresses.
- We expect that taint colors from tainted bytes are frequently combined during program execution. Therefore, we will support an arbitrary number of color combinations, improving the typical bitmap implementation.

Considering the goals described previously, the taint storage of our engine follows the architecture shown in Figure 4. As it can be observed, we dispose of two main structures, one for storing memory colors and another for register colors. In the case of memory tainting, we store in an unordered map pairs of data corresponding to each individual tainted memory address and the color that it has been tainted with. This map is modified at runtime, so that addresses not contained in the map are assumed not to be tainted.

On the other hand, we will track the taint in each of the registers of the process by means of a vector that acts as a shadow processor. In this vector, we will contain one element for each byte of each register (including *rax*, *rbx*, *rcx*, *rdx*, *r8 ... r15*, *rsi*, *rdi*, *rsp* and *rbp*). Each element will hold the color with which a specific byte of that register has been tainted. Note that a value of 0 means that the register byte is not tainted. Also, support for the x86 architecture is guaranteed by, at runtime, ignoring half the bytes for each register. We follow the same technique when instrumenting a program that makes use of, for example, 8-byte registers like *al*, by only taking the last 8 bytes in the vector corresponding to register *rax*.

Multi-color support. Although the previous architecture already allows us to store our taint data, a multi-color engine requires additional data structures for taint colors. During the program execution, taint colors will be generated and combined to create new ones. These combinations, known as color mixes in our engine, will need to be stored so that we can track back which original colors were combined to generate a certain derivate color.

The tainting module features the color mixes functionality by means of two additional data structures. The current version of *TaintBlade* supports binary color mixes, meaning that any color is either original or the combination of two other colors. As we will describe, the color mixes data structures will need to be frequently accessed for two main purposes:

- Knowing whether a certain color has been already generated as a result of the mix of two other colors.
- Getting the colors that were combined to generate a certain derivate color.

Therefore, we have developed the data structures that are detailed in Figure 5, consisting on two maps. The first map stores, for each entry, a color that was generated as a mix of two other colors, and the colors used in the mix. This is useful since we frequently need to access whether a color mix has been generated already. In turn, the second map stores, for each color, which colors it has been mixed with, and the result of the mix. This is particularly needed for the case of getting the result of the mix of two specific colors, since without an additional map we would need to traverse the whole first color mix map.

Taint sources. Once considered how taint colors are stored in the tainting module, we will discuss how these colors are generated in the first place. *TaintBlade*'s tainting module follows the architecture described in section II-C, by specifying taint sources which generate the taint colors at specific registers and memory bytes.

In *TaintBlade*, every taint source is a routine. A routine declared as a taint source will perform the following actions:

1) Using the instrumentation module, hook the routine before its execution and take the arguments with which it was called.
2) Interpret the arguments received, casting them to the data structure that we know corresponds to them, previously hardcoded.
3) Using the instrumentation module, hook the routine after its execution and the arguments with which the function returns.
4) Again, cast the value of the arguments at return time to the data structure that has been hardcoded.
5) Depending on the routine, taint with different colors specific bytes of the arguments that have been either received or returned.

Currently, *TaintBlade* is focused on working with network protocols and, therefore, the taint sources we have defined correspond to those routines most commonly used for receiving network information. Table III details the taint sources currently implemented and which bytes get tainted when

they are triggered. Note, however, that these could be easily extended in the future to support any arbitrary routine.

Taint sinks. Just like we define program points at which to generate taint colors, *TaintBlade* also incorporates a series of routines acting as taint sinks, at which we are interested to know whether any tainted data is received via the passed arguments. In this initial version, we have incorporated three routines, detailed at Table III, that are relevant for our protocol reverse engineering task and perform certain actions when triggered:

1) CreateProcessA and CreateProcessW will receive a command to be executed. By defining them as taint sinks, we will be alerted whenever a tainted byte (that comes from the network) happens to be one of these commands.
2) In Windows, due to historical reasons, many API calls are available either with ANSI strings in the arguments (taking *char**) or with UNICODE "wide" strings (as *wchar_t**). MultiByteToWideChar is a function that allows us to convert strings between these two types, and it can thus be commonly found in programs. By defining this routine as taint sink, we are able not only to detect whether it receives tainted data, but also to transfer this data to the new, converted string, therefore ensuring we do not incur on undertainting.

Whenever some taint color is found to reach a taint sink, that taint color is logged in an internal vector, so that it can later be used when reversing the protocol.

Taint propagation policy. Once considered how taint colors are stored in the tainting module, we will discuss how this data is propagated during the program execution. The goal of the taint propagation policy is to define a set of rules that specify how taint colors are moved across memory and registers depending on the instructions being executed. There exist different rules, each corresponding to one of the instruction types we instrumented using the instrumentation module and that we detailed at Table I. Note, however, that not all the instructions types incur in moving data, therefore only those operating with data will have an attached taint rule.

As detailed in the table, every taint rule results on either tainting, untainting or mixing the colors of one or more operands:

- If an operand gets tainted by another, it means that the colors of the destination operand get overwritten by those from the source operand.
- If an operand gets untainted, it means that its colors get removed, all set to 0.
- If an operand gets mixed with another, it means that the colors of the destination operand get mixed, one by one, with those from the source operand.

Mixing colors consists on the following steps:

1) Take the two colors to mix. Check in the reverse color mix map if that combination already exists.
   a) If the color exists in the map, it means that the color mix has been produced before. Therefore, the result of the mix is the previously generated mix color.
   b) If the color does not exist in the map, this is a new mix. Therefore, the result of the mix is equal to the latest color mix produced + 1. (e.g. if the last mix was color 5, then the new mix is color 6).
2) Take the resulting mix color and log the mix in the mix maps.

Indirect tainting. During the program instrumentation, it is possible to find assembly operations (e.g. *LEA*s) that operate on registers pointing to memory rather than the actual memory values. While our taint rules do not account for pointers to tainted memory (as values stored in memory do not propagate their taint colors to memory pointers), it is needed to identify instances where pointers to tainted data are manipulated. This identification allows us to detect pointer fields, which by defined to be offsets to pointers to tainted memory addresses.

D. Heuristics module

The previous two modules have instrumented the instructions and routines to extract their arguments and operands, as well as generated and propagated the taint colors during the program execution. However, in order to achieve our reverse engineering task, we must be able to interpret the list of instrumented instructions and their associated taint colors, so that we can eventually extract the underlying operations that the program is performing with those instructions. The heuristics module does exactly that. Its goal is to match assembly instructions and taint data with the high-level operations the program is performing. In order to achieve this matching, we perform the following process:

Normalization. Firstly, *TaintBlade* normalizes the gathered instrumentation data by encapsulating the instruction information extracted by the instrumentation module (the type of instruction, the value of the operands...) and the taint colors of each operand indicated by the tainting module into an standard structure known as a 'RevAtom'. This RevAtom matches each operand values with the taint colors they have, and offer an abstraction so that the heuristic module can operate with all data independently on the instruction type. In essence, a RevAtom contains the following data:

1) The instruction type.
2) For every operand in the instruction, it contains:
   a) The value of the operand.
   b) The taint color of the operand.
3) Other optional data. This depends on the instruction type. For instance, for *cmp* operations, we store the value of the RFLAGS register after the instruction execution, so that we know if the comparison was successful. All optional data is gathered via the instrumentation module.

Once a RevAtom is generated, *TaintBlade* will check whether there exists any operand in the RevAtom that holds a taint color. If every operand is untainted, it means that, at a high-level, that instruction does not take part in the

TABLE III: Implemented taint sources and taint sinks in TaintBlade

	Routine name	DLL	Arch	Arguments	Taint actions
Taint sources	Recv[a]	ws2_32.dll wsock32.dll	x86 x86_64	SOCKET s; char* buf; int len; int flags;	At exit, taints as many bytes from argument *buf* as indicated in the return value.
	InternetReadFile[b]	wininet.dll	x86 x86_64	LPVOID hFile; LPVOID lpBuffer; DWORD dwNumberOfBytesToRead; LPDWORD lpdwNumberOfBytesRead;	At exit, taints as many bytes from the argument *lpBuffer* as indicated in argument *lpdwNumberOfBytesRead*.
Taint sinks	CreateProcessA[c]	kernel32.dll	x86 x86_64	LPCSTR lpApplicationName; LPSTR lpCommandLine; LPSECURITY_ATTRIBUTES lpProcessAttributes; (+7 other arguments)	Before executing the function, it checks whether any byte from argument *lpApplicationName* is tainted.
	CreateProcessW[d]	kernel32.dll	x86 x86_64	LPCWSTR lpApplicationName; LPWSTR lpCommandLine; LPSECURITY_ATTRIBUTES lpProcessAttributes; (+7 other arguments)	Before executing the function, it checks whether any byte from argument *lpApplicationName* is tainted.
	MultiByteToWideChar[e]	kernel32.dll	x86 x86_64	UINT CodePage; DWORD dwFlags; LPCCH lpMultiByteStr; int cbMultiByte; LPWSTR lpWideCharStr; int cchWideChar;	After executing the function, take a byte *i* from *lpMultiByteStr* and, if tainted, taint with the same color the byte *i* at *lpWideChar*, for a maximum of bytes indicated in the return value.

[a] Routine and arguments defined following https://learn.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-recv
[b] Routine and arguments defined following https://learn.microsoft.com/en-us/windows/win32/api/wininet/nf-wininet-internetreadfile
[c] Routine and arguments defined following https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa
[d] Routine and arguments defined following https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw
[e] Routine and arguments defined following https://learn.microsoft.com/en-us/windows/win32/api/stringapiset/nf-stringapiset-multibytetowidechar

TABLE IV: Taint rules for each instruction type

Instruction type	Taint rule
Arithmetic operations	Mix destination operand colors using colors of source operand.
Logical operations	If the operands are the same, untaint the operands. Otherwise, mix destination operand colors using colors of source operand.
Shift operations	Calculates the number of bytes shifted. Moves every color of every byte of the register to the right (from MSB to LSB). If the color goes out of bounds, the color is lost.
Data moving operations	Untaint the destination operand. Then, taint the destination operand with the colors from the source operand.
Address moving operations	Take colors of reg. leaBase and reg. leaIndex. Mix the colors with those of destination reg. Calculate indirect tainting.
String manipulation operations	Take colors from memory being read. Mix them with colors of register *al*, *ax*, *eax* or *rax* depending on instruction length. Also, mix them with colors of register *di*, *edi* or *rdi* depending on instruction length.

management of tainted data, and thus has no relation to the underlying protocol of the program. Therefore, if every operand is untainted, the RevAtom is discarded. In the case that some operand is tainted, then we are interested in examining that instruction further. In this case, the tool will introduce the RevAtom in a centralized vector known as the 'RevLog'.

Heuristic analysis Once we have a new RevAtom in the RevLog vector, *TaintBlade* will try to match the RevAtoms in the RevLog to a set of heuristics. A heuristic is an instruction, or a set of instructions, that define a high-level operation happening in a program. For instance, a *cmp* instruction where some of the operands are tainted signals that the program may be looking for some value within the tainted data. In this case, we would look in the RevLog for RevAtoms containing a *cmp* instruction and some tainted operand.

As we explained in the introduction, our goal is to detect keywords, delimiters and pointer fields in the protocol. In order to do this, we have developed the heuristics detailed on Table V. The table highlights two distinct heuristics: the *comparison operation* heuristic, which corresponds to the program performing a comparison against some byte, and the *pointer operation* heuristic, which denotes a scenario where a value is added to a pointer, causing it to point to another address within the tainted buffer.

In order to match the RevAtoms to the heuristics, the

TABLE V: Implemented heuristics in TaintBlade

Heuristic	Instructions
Comparison operation	CMP(mem, imm), where mem is tainted.
	CMP(reg1, reg2), where reg1 is tainted.
	CMP(reg1, reg2), where reg2 is tainted.
	CMP(mem, reg), where mem is tainted.
	CMP(reg, mem), where reg is tainted.
	CMP(reg, imm), where reg is tainted.
Pointer operation	LEA(memDest, [leaBase+(leaIndex*leaScale)+leaDis]), where an indirect taint is detected.

module runs a simple search algorithm in the RevLog. This algorithm is a loop that scans the RevLog, starting from the last inserted instruction, and checks whether there is any combination of instructions that match one of the heuristics. Once an heuristic is found, the module extracts it and inserts it in an internal vector.

Note that, although this version of *TaintBlade* does not incorporate heuristics with multiple instructions, the algorithm supports them already.

E. Protocol reversing module

As we described, during the program execution *TaintBlade* will accumulate all found heuristics in an internal vector. The final step is to interpret the operations described in the heuristics, grouping or discarding them, so that we are able to complete the message format inference of the underlying protocol. This process happens only once the program has finished.

In *TaintBlade* we follow the field classification we described in Section II-A. In order to reverse the protocol, a custom algorithm is used. This algorithm receives the vector of found heuristics, the taint sources and the taint sinks information. Then, by considering the chronological order at which the operations occured and the nature of the encapsulated instructions, it produces a list of protocol messages and fields. We will now detail how each type of element is reversed.

Protocol messages. Protocol messages are identified by considering taint sources data. As we show in Table III, each taint source taints a buffer in memory. Each buffer then corresponds to one message in our protocol.

Delimiters and keywords. The detection of these fields requires to consider the gathered comparison operations (found by the heuristics module) that correspond to a given protocol message. Moreover, we need to take into account the temporal location (the chronological order at which the comparisons were produced) and the message location (the byte offset within the message at which the comparisons are performed) across all the comparison operations. This is key since the program may concatenate multiple sequential comparisons when it needs to check for a long keyword, whilst we will need to consider repeated comparisons in order to find delimiters.

The first step needed is to organize the comparison operations considering their temporal and message location. For this, we will take each comparison operation and extract every individual byte comparison (since each comparison may be made of up to 8 bytes). Every byte comparison is then put in a list.

Next, we will assign an index to each byte comparison in the list, referred to as the *heuristic level*. The purpose of this index is to indicate the temporal and message location relationship among byte comparisons. In essence, byte comparisons sharing the same heuristic level would indicate that they occurred sequentially in time and involved the comparison of sequential bytes in the protocol message. The simplified pseudocode is shown at Appendix VI-A.

Once we have assgined an heuristic level to each byte comparison, we can then construct the delimiters and keywords in the protocol. We will produce the following classification:

- We classify a group of byte comparisons as a delimiter when they share heuristic level, and the comparison involves checking the same byte value in every byte. Specifically, we will detect a *failed delimiter* when each byte comparison is failed, and a *successful delimiter* when each byte comparison is failed with the only exception of the last one (where the comparison succeeded).
- A group of byte comparisons is classified as a keyword when they share a heuristic level and they were not classified as a delimiter previously. Note that, by using the heuristic level mechanism, we ensure that long keywords are detected as such, since all their bytes share the same heuristic level in the list.

Note that, sometimes, it is possible to find keywords of a single byte, where the comparison failed, and which are unable to be joined to another keyword. Due to the nature of dynamic analysis, this is an event that happens when the program checks a keyword byte by byte instead of using, for instance, a common routine like *strcmp()*. In this cases, *TaintBlade* marks the byte as a keyword but highlights the fact that it is isolated so that the user has the chance of investigating it further if wanted.

Pointer field. The detection of pointer fields is managed through the traversal of each pointer operation within the heuristics vector. Since every pointer operation represents an atomic event where the program acquires an offset from the protocol message, we can directly translate them into pointer fields once their corresponding message is determined. For each pointer operation found, *TaintBlade* analyzes each pointer operation, determining the specific byte within the message that serves as the source and identifying the byte (and its associated message) that the source byte is pointing to.

Taint leads. Taint leads are optional, semantic information that applies to a single byte in a protocol message. These elements are not generated by the reversing algorithm using the gathered heuristics, but rather they are filled using the information about which colors were detected to reach certain

taint sinks. A taint lead allows the protocol to enrich the message inference with data from the taint analysis, so that a byte at a fixed-length field or variable-length field is assigned with the lead. This semantic information includes the routine at which the data was used, the argument at which it was passed to the function and the offset inside the argument at which the specific byte was found.

Variable-length field We detect variable-length fields by inspecting the memory addresses pointed by pointer fields. If a byte is pointed by a pointer field, and it is not yet assigned either to a delimiter or a keyword, then it gets assigned as the starting byte of a variable-length field. Furthermore, any sequentially following byte in the message will be included in this group until a byte is found that has already been classified as a different type of field.

Finally, we indicate whether any byte in the field was used in some relevant routine of the program by means of the taint leads.

Fixed-length field A fixed-length field is detected if a group of bytes is assigned semantic information (by means of a taint lead) and they do not already belong to any other field type.

F. Tracing module

This is an auxiliary module that allows for instrumenting any routine specified by the user. It makes use of the instrumentation module to gather all operands of the routine right after it is called, and before it returns. The user is responsible for selecting which routines to trace, and all information is dumped to the output files and DB, and shown at the GUI.

Additionally, for every traced argument, the module attempts to access its string value (if any). For this, it reads the memory pointed by the argument and tries to interpret it either to a *char** string or a Unicode "wide" string (with *wchar_t**).

G. Noper module

This is an auxiliary module that helps to instrument programs loaded with stealth mechanisms, which is particularly common in malware. It makes use of Pin's instrumentation to allow the user to skip specific code sections. For this, the user introduces a range of RVAs that the program will jump over when the instruction reaches that point.

Additionally, it also allows for modifying, after jumping over the specified fields, the values of the registers. This is particularly useful to avoid execution specific routines (such as the ones checking whether a program is being debugger) and then modifying the return value, so that a potentially malicious program does not modify its behavious once it detects the sandbox.

H. TaintBlade GUI

The final component of the *TaintBlade* framework is the GUI. It is an interface made using Qt6 that allows the user to easily access and navigate the data produced by the*TaintBlade* Pintool, plus it adds additional features on top of it.

Figure 6 shows the GUI after running a sample program that features all type of fields. In essence, this program receives an array of bytes from the network and checks multiple delimiters and fields, then executing a command using some bytes in the message as the program name. We will not go into more detail since are only interested in the visualization, but the details and the program being instrumented can be found in the project repository at */samples/tcp_client.cpp*. We will also later go in depth about how to use the GUI with a real malware sample during the evaluation section.

The GUI shows a toolbar with menus for customizing the analysis and selecting options. Going from left to right:

- A menu for selecting the binary to instrument.
- A menu for selecting options for the instrumentation, such as a directory where to store output files.
- A menu for starting the instrumentation, and another for stopping it.
- A menu for selecting the taint sources to use from those available at the tainting module.
- A menu for selecting the routines to trace using the tracing module.
- A menu for selecting the RVAs to skip using the noper module.
- A menu for selecting the DLLs to include in the scope of the instrumentation.

Regarding the main screen, it is made of resizable windows, each displaying a different functionality. The following is a summary of the windows' functionality following the labels we have drawn:

1) The **process window**. This window indicates which processes where executed by the instrumented binary, since many times a binary delegates functionalities into child processes.
2) The **protocol fields window**. This window displays the different messages and fields inferred during the program execution. It lists all buffers (messages) detected, and clicking on them shows the fields detected for that message.
3) The **field details window**. This window shows information about the comparisons that delimiters and fields are made of, and information about pointer fields (such as the byte they point to).
4) The **message visualization window**. This video shows the bytes that each message is made of. Bytes belonging to delimiters or keywords are grouped and colored. Additionally, it also allows for displaying the semantic meaning of fields (e.g. which bytes reach taint sinks).
5) The **taint events windows**. This window shows which routines are responsible of generating, mixing and moving the taint colors during the execution. It displays the image and RVA of every instruction reached by taint data. Additionally, it allows for displaying which instructions belonging to scoped images are responsible for jumping or calling external routines that manage taint data (e.g. visualizing where is the *call* instruction that jumps to a taint source routine).
6) The **traced routines window**. This window displays

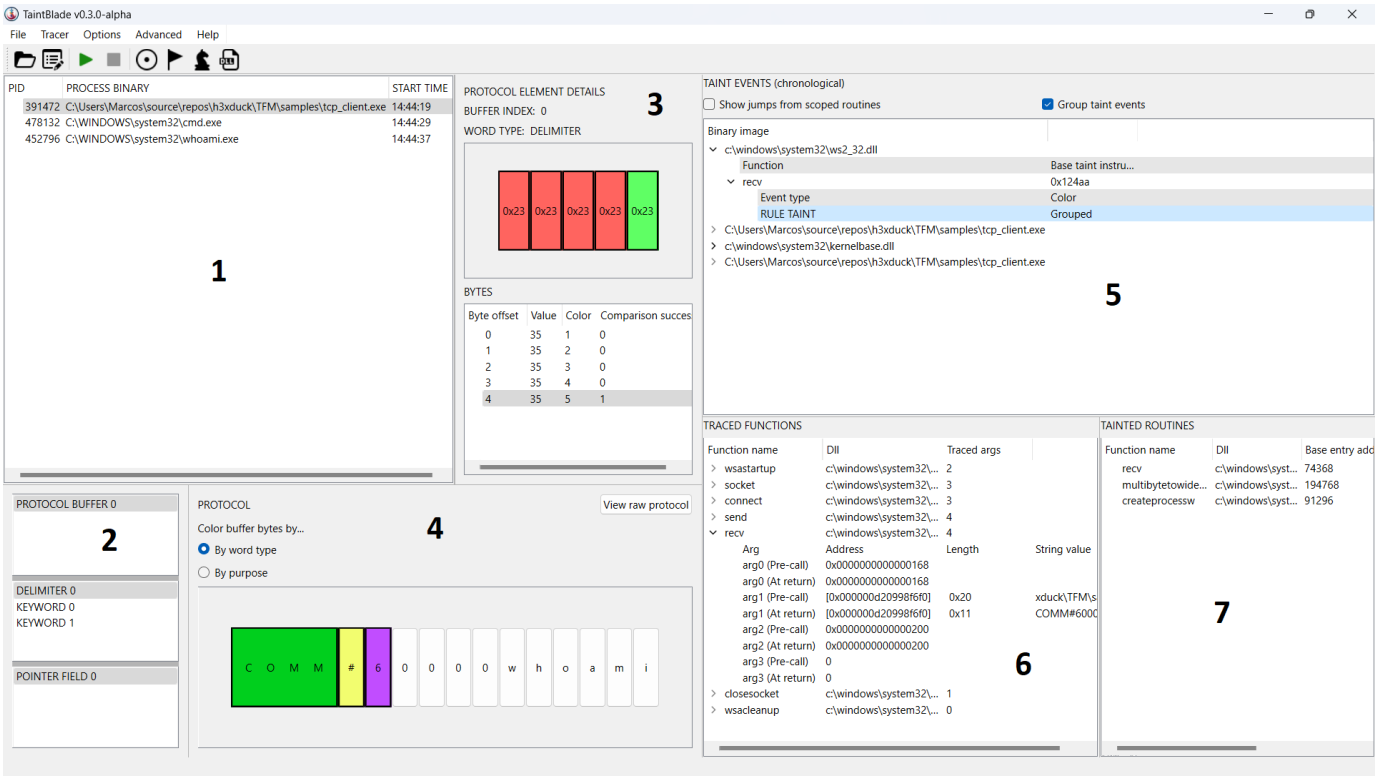

Fig. 6: Screenshot of the GUI after instrumenting a sample program.

which routines selected by the user where executed, along with their arguments at the time of calling and return. Also, incorporates the string value of the argument when possible.

7) The **tainted routines window**. This window displays the taint sources and taint sinks that have been involved in the taint analysis.

Each window incorporates multiple functionalities activated when left and right-clicking on the displayed elements. The windows do not work separately, but rather the selected data modifies the visualization at other windows. For example, selecting a field at the protocol fields window highlights the corresponding bytes at the message visualization window and changes the displayed field at the field details window.

We encourage to visit the project repository [22] to learn about all the features of *TaintBlade*'s GUI.

I. Code availability

All the source code belonging to the *TaintBlade* framework can be visitied publicly at the GitHub repository *https://github.com/h3xduck/TaintBlade* [22]. The most important folders and files of this repository are described in Table VI.

TaintBlade has been tested with Intel Pin v3.25. Currently, it is only available for Windows systems.

IV. Evaluation

This section evaluates the *TaintBlade* framework by using it in a real scenario of analyzing a malware sample.

TABLE VI: Relevant directories at the TaintBlade repository.

Directory	Description
src/PinTracer	Source code of *TaintBlade*'s Pintool.
src/PinTracer/engine	Instrumentation of executed instructions. Includes taint propagation routines.
src/PinTracer/reversing	Source code of the heuristics and protocol reversing modules.
src/PinTracer/taint	Source code of tainting module, includes taint sources, sinks and taint storage.
src/external	A distribution of Intel Pin for Windows.
src/GUI/PinTracerUI	TaintBlade's GUI source code.
src/samples	Sample programs, can be used for testing TaintBlade.
src/PinTracer/x64 /Release/PinTracer.dll	TaintBlade's Pintool compiled executable.
src/GUI/PinTracerUI/x64 /Debug/PinTracerUI.exe	TaintBlade's GUI compiled executable.

The malware sample we will be using is a Brbbot trojan for 64-bit Windows systems. It is packed with UPX [23] and gains persistence using the Windows registry. Once activated, the malicious process remains in the background, connecting to a C2 server periodically. This communication is done via HTTP GET requests (from the client to the C2 server) and their HTTP responses (from the C2 server to the client). The malware has the capability of executing the commands sent from the C2 server. The SHA-256 hash of the sample is *f9227a44ea25a7ee8148e2d0532b14bb640f6dc52cb5b22a9f4fa7fa037417fa*.

The goals of this evaluation are the following:

1) Accurately infer the protocol message format of the malware sample, without needing access to its source code. We will not have any information about the sample functionality prior to our analysis. Additionally, we will not contact the real C2 server, but rather we will explain how the protocol can be extracted via a process of manual fuzzing. This more closely resembles a real scenario, since the C2 server may not be available.
2) In addition to the syntactic information, we aim to infer the semantic information of the bytes received during the C2 communication. We are particularly interested in learning how the trojan executes commands sent from the C2 server.
3) Get an accurate visualization of the reversed protocol format in the GUI. This includes getting knowledge about the routines that are used during the malware execution and which of them operate with tainted data.

Experimental setting. The test environment that will be used during this evaluation consists on two virtual machines running under Oracle VM VirtualBox [24]:

- The first machine is a Windows 10 system. This machine has been modified using Flare-VM's scripts [25]. It will be in charge of running the malware sample under *TaintBlade*'s instrumentation. Here, we will also launch the GUI to visualize the results.
- The second machine is a REMnux Linux system [26]. In this machine, we will make use of the INetSim tool for simulating the C2 server.

Both machines will be connected via a Host-only network. This ensures that the malware can contact the C2 simulator but without using a physical networking interface, so that it cannot communicate outside of the sandbox.

Compilation. The first step is always to compile the two *TaintBlade* components: the Pintool and the GUI. The two projects can be found under the directories specified at Table VI.

- In the case of the Pintool, the solution must be opened and compiled using Visual Studio. The compilation has been tested successfully in VS 2022, but it may work for other versions.
- In the case of the GUI, it requires Qt 6.5.1. It is prepared to be compiled as a Visual Studio project. Again, VS 2022 is the only tested version.

Once both components are compiled, we can find the Pintool DLL under the directories indicated at Table VI.

First instrumentation. Once we have compiled both the Pintool and the GUI, we are ready to start the instrumentation. We will now describe the steps that must be followed to instrument the sample using the GUI. Note that, alternatively, a user can also use *TaintBlade*'s Pintool directly through the command line. Check our project repository for how to do it and the program flags available. Also, we will be leaving screenshots for most of the steps in Appendix VI-A.

In the first instrumentation, we will leave INetSim in the default configuration, that is, it will catch any request sent by the malware and answer with generic data. Next, we launch the GUI by executing the GUI binary.

The first step for using the GUI is selecting the malware binary to instrument, the location of *TaintBlade*'s Pintool and a directory where to store the output files (Figure 9).

Once set, we select the taint sources we want to use in our instrumentation. Since we do not know yet which ones the program uses, we leave all of the marked (Figure 10). Note that it is also possible to investigate which are the routines being executed by the malware using the tracing module, so that we can ensure that the APIs it uses are supported in *TaintBlade* as taint sources (Figure 11).

Once we have selected the taint sources, we are ready to run the malware. We click the play button at the GUI and wait. At this point, the Pintool will start instrumenting the binary, and we will see that the GUI shows the process currently running in the process window.

In many malware samples, the malicious process tends to run indefinitely. This is logical since, once executed, the malware aims to operate as long as possible, and may event attempt to get persistence. Therefore, we must terminate the process using the stop button in the GUI. The moment at which we decide to terminate the program depends on the analyst, but usually this is when we detect that the malware has started to communicate throught the network. In our case, we detect that the program has issued a HTTP GET request to <C2domain>.php and received a response (from INetSim). Note that it is also possible to set a timeout to automatically stop the program execution after a certain time, if using the CLI. You can access all the program flags available at our project repository .

Once we stop the execution, we will be able to visualize the messages gathered and the fields found by the Pintool using message format inference. By clicking into the process at the process window, the protocol fields window will display the fields detected, and the message visualization window will draw the bytes and fields that conform the message. Also, the taint events and the traced routines window will start showing the used taint sources/sinks and the routines that have been traced respectively (Figure 7).

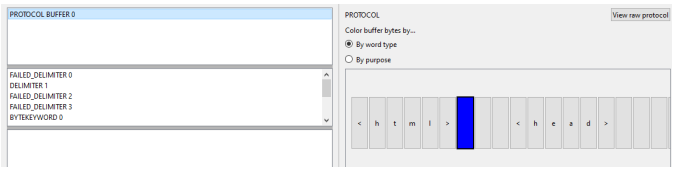

Fig. 7: Visualization of gathered messages and fields after stopping the execution.

By clicking on each field at the protocol fields window, we can display the details of each field in the message. We can also click the raw protocol button to access the raw data gathered by the tool (Figure 12). By using any of these methods, we will find that there the first keyword that the program checked was the following (Figure 13):

BYTEKEYWORD | STARTINDEX:0 | ENDINDEX:0 Byte 0: 63 (as char: c) | Comparison result: 0

This indicates that the program was looking at position 0 of the message for the character 'c', and failed to find it. Due to the nature of dynamic analysis, it may happen that this is the first byte of a longer keyword which the program is checking one letter at a time.

Protocol fuzzing Taking into consideration the above, we decide to run the protocol inference a second time. This is a similar process as trying to 'fuzz' the protocol format, since we will be trying to enter the keywords that the malware is looking for to see if something changes.

For this, we will modify the data that INetSim sends to the malware, so that it starts with the character 'c'. Then, we repeat the same process as before. The result is that, this time, we can see that the program checked for character 'e' at the next byte after the comparison with character 'c' succeded (Figure 13):

KEYWORD | STARTINDEX:0 | ENDINDEX:1 Byte 0: 63 (as char: c) | Comparison result: 1 Byte 1: 65 (as char: e) | Comparison result: 0

Therefore, it is clear that there is a hidden keyword. We will now proceed to repeat the process as many times as characters appear in the keyword. For each new byte that the program fails to find, we will add it into the message sent by INetSim.

Final protocol format. After we repeat the fuzzing process for two more bytes, we eventually reach the following delimiter (Figure 15):

KEYWORD | STARTINDEX:0 | ENDINDEX:3 Byte 0: 63 (as char: c) | Comparison result: 1 Byte 1: 65 (as char: e) | Comparison result: 1 Byte 2: 78 (as char: x) | Comparison result: 1 Byte 3: 65 (as char: e) | Comparison result: 1

At this point, using the GUI, we can observe that there is a new variable-length field with associated semantic data. By right-clicking on it, we can highlight in the taint routines window which taint sink has received those bytes of the message. We can see that, up until the newline delimiter, all bytes will be sent into the routine CreateProcessA (Figure 16).

Therefore we now know that the message follows the following format:

cexe <COMMAND><newline>

We can check that this is the case by configuring INetSim to send the following string to the malware:

cexe c:\windows\system32\calc.exe

Once we instrument the malware process sending that message, we will see that a calculator program is run. Also, we can see in the GUI the final form of the message format, with all successfully checked keywords and delimiters (Figure 17) and the variable-length field whose semantic data indicates that it was sent to the routine CreateProcessA, as shown in Figure 8.

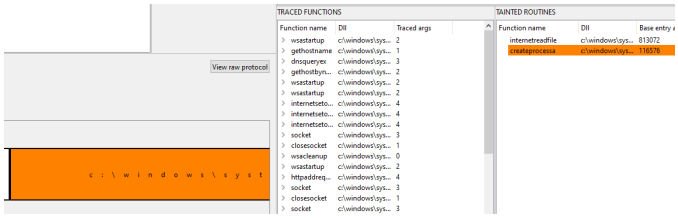

Fig. 8: Protocol message visualization window, showing the semantic information of a variable-length field.

V. Related work

As we discussed in previous sections, protocol reverse engineering is a mature field. Since the 2000s, there has been multiple research papers exploring how to reverse protocols with different static and dynamic techniques. However, as we indicated, most of these tools do not have their code available. Usually, they are mere proof of concepts, instead of trying to appeal to general public by being usable and accessible. In this section, we will compare *TaintBlade* with two works: Netzob, which is an open-source project known to be usable, and Polyglot, which although mostly closed, it is one of the most complete protocol reverse engineering frameworks.

Netzob [12] is the largest open-source protocol reverse engineering project. It performs message format inference as a PF, and also extracts the protocol grammar as a PFSM. For this, it works with static analysis techniques, such as clustering [27], in order to partition the message into fields. It does not, however, include any semantic information into the fields (since for this dynamic analysis is needed). Netzob also includes a fuzzing system that allows for discovering the different behavior of a program when receiving different messages.

TaintBlade does not infer the protocol grammar as Netzob but, in turn, we achieve a more accurate message format inference with our dynamic techniques, getting semantic information. On the other hand, static analysis is much faster than *TaintBlade*'s instrumentation. Regarding the fuzzing system, it is clearly a needed feature when performing protocol reversing (we did fuzzing manually at the evaluation section), and is considered as future work that will eventually be added to our tool. Finally, *TaintBlade* has a GUI and a CLI, whilst Netzob is CLI-only.

Polyglot [5] is one of the most well-known protocol reverse engineering research works. It has been followed by other systems from the same authors such as Dispatcher [28], which is a protocol reversing tool for sent messages (instead of only received messages). Overall, it is a very mature project with a large group of researchers behind, and many of the ideas of *TaintBlade* are based on it.

Polyglot uses dynamic analysis techniques, that includes a custom dynamic analysis platform known as TEMU, based on the emulator QEMU [29], that achieves instruction-level instrumentation. However, it virtualizes the whole system instead of instrumenting a single binary. In turn, *TaintBlade* achieves instruction-level instrumentation using Intel Pin and

a custom multi-color taint engine. Polyglot features message format and grammar inference, whilst *TaintBlade* is only focused on formats. Finally, Polyglot does not have a user interface, whilst we believe that the *TaintBlade*'s GUI and the helper modules (like the noper module) makes TaintBlade a more usable solution. Additionally, the majority of its code is not open to the public (some components are, such as TEMU).

VI. Conclusion

In this work, we presented a practical and powerful solution for reverse engineering protocols. Our framework, based on Intel Pin, enables comprehensive instrumentation of various programs on Windows systems across multiple architectures. By leveraging this tool, we successfully inferred the message format of a command and control protocol used by a malware trojan, even without accessing its C2 server. Additionally, we uncovered the mechanism through which the C2 server transmits executable commands to the trojan.

Our solution comprises two key components: the Pintool and the GUI. The Pintool facilitates precise program instrumentation and extensive data extraction, while the GUI provides a user-friendly interface for seamless data navigation and visualization.

In the case of the Pintool, it is a modular tool with features a unique multi-color tainting engine, which we believe to be a significant contribution. As far as we know, there does not exist any other multi-color tainting framework for 64 bits machines. Moreover, the Pintool is designed to be more than just a black-box tool, since it incorporates additional modules that facilitate the work of an analyst. For instance, the routine tracing module allows easy access to the arguments of any routine, and the noper module enables the skipping of arbitrary code sections, which is particularly useful for malware analysis. The remaining modules (instrumentation, tainting, heuristics, and protocol reversing) enable the inference of six different types of fields in the message format: delimiters, keywords, pointer fields, variable-length fields, and fixed-length fields. These modules also include semantic data, informing the user of the purpose of the field's purpose in the program, such as its use as a command.

Regarding the GUI, we have developed a production-ready tool that seamlessly integrates with the Pintool. It allows users to easily visualize and interact with the data, providing a cohesive and intuitive display.

In summary, our work is a great example of how to use dynamic analysis techniques for protocol reversing, whilst also serving as a useful tool for analyzing malware and non-malicious programs.

A. Future work & limitations

Although *TaintBlade* incorporates many features, during our research we identified a series of capabilities and other functionalities which we believe would be a great addition to the work presented here:

- **Support for encrypted and compressed protocols.** We dedicated a significant amount of time to the research on how to analyze protocols where the messages are not sent in the clear. Even in the case of keywords, where the program will eventually check the clear-text value of the message, an accurate system to detect when the message has been decrypted is needed. Some options we discussed included detecting encryption routines by counting the number of loops and logical operations (since these are very common in encryption and decryption), but this feature was left out of scope due to not being finished at the time of the project deadline.
- **Incorporation of a fuzzing system.** As we noticed while evaluating the tool with malware samples, an automated fuzzing system that analyzes the program behaviour upon different inputs would greatly enhance the analysis with *TaintBlade*.
- **Support for Linux.** Intel Pin is multi-platform, and therefore *TaintBlade* could easily support Linux systems. The main change that would need to be done is the addition of system calls as taint sources and taint sinks.

References

[1] B. D. Sija, Y.-H. Goo, K.-S. Shim, H. Hasanova, and M.-S. Kim, "A Survey of Automatic Protocol Reverse Engineering Approaches, Methods, and Tools on the Inputs and Outputs View," Security and Communication Networks, vol. 2018. Hindawi Limited, pp. 1–17, 2018. doi: 10.1155/2018/8370341. Available: http://dx.doi.org/10.1155/2018/8370341

[2] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," Journal of Computer Virology and Hacking Techniques, vol. 14, no. 1. Springer Science and Business Media LLC, pp. 53–68, Jan. 25, 2017. doi: 10.1007/s11416-016-0289-8. Available: http://dx.doi.org/10.1007/s11416-016-0289-8

[3] Y. Huang, H. Shu, F. Kang, and Y. Guang, "Protocol Reverse-Engineering Methods and Tools: A Survey," Computer Communications, vol. 182. Elsevier BV, pp. 238–254, Jan. 2022. doi: 10.1016/j.comcom.2021.11.009. Available: http://dx.doi.org/10.1016/j.comcom.2021.11.009

[4] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," Journal of Computer Virology and Hacking Techniques, vol. 14, no. 1. Springer Science and Business Media LLC, pp. 53–68, Jan. 25, 2017. doi: 10.1007/s11416-016-0289-8. Available: http://dx.doi.org/10.1007/s11416-016-0289-8

[5] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot," Proceedings of the 14th ACM conference on Computer and communications security. ACM, Oct. 28, 2007. doi: 10.1145/1315245.1315286. Available: http://dx.doi.org/10.1145/1315245.1315286

[6] "BitBlaze Binary Analysis Platform Release Information." Available: http://bitblaze.cs.berkeley.edu/release/index.html

[7] Behzad-A, "GitHub - behzad-a/Dytan: Dytan Taint Analysis Framework on Linux 64-bit," GitHub. Available: https://github.com/behzad-a/Dytan

[8] Jasantunes, "GitHub - jasantunes/reverx: A protocol reverse engineer tool written in java.," GitHub. Available: https://github.com/jasantunes/reverx

[9] Tammok, "GitHub - tammok/PRISMA: Protocol Inspection and State Machine Analysis," GitHub. Available: https://github.com/tammok/PRISMA/

[10] Hgascon, "GitHub - hgascon/pulsar: Protocol Learning and Stateful Fuzzing," GitHub. Available: https://github.com/hgascon/pulsar

[11] Vs-Uulm, "GitHub - vs-uulm/nemesys: NEtwork MEssage Syntax analysYS (WOOT 2018) and NEtwork MEssage TYpe identification by aLignment (INFOCOM 2020)," GitHub. Available: https://github.com/vs-uulm/nemesys

[12] Netzob, "GitHub - netzob/netzob: Netzob: Protocol Reverse Engineering, Modeling and Fuzzing," GitHub. Available: https://github.com/netzob/netzob

[13] J. Clause, W. Li, and A. Orso, "Dytan," Proceedings of the 2007 international symposium on Software testing and analysis. ACM, Jul. 09, 2007. doi: 10.1145/1273463.1273490. Available: http://dx.doi.org/10.1145/1273463.1273490
[14] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft," ACM SIGPLAN Notices, vol. 47, no. 7. Association for Computing Machinery (ACM), pp. 121–132, Mar. 03, 2012. doi: 10.1145/2365864.2151042. Available: http://dx.doi.org/10.1145/2365864.2151042
[15] AngoraFuzzer, "GitHub - AngoraFuzzer/libdft64: libdft for Intel Pin 3.x and 64 bit platform. (Dynamic taint tracking, taint analysis)," GitHub. Available: https://github.com/AngoraFuzzer/libdft64
[16] JonathanSalwan, "GitHub - JonathanSalwan/Triton: Triton is a dynamic binary analysis library. Build your own program analysis tools, automate your reverse engineering, perform software verification or just emulate code.," GitHub. Available: https://github.com/JonathanSalwan/Triton
[17] D. Andriesse, Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. No Starch Press, 2018.
[18] AMD, "AMD64 Architecture Programmer's Manual Volume 2: System Programming." Available: https://www.amd.com/system/files/TechDocs/24593.pdf, Page 65
[19] "Pin - A Dynamic Binary Instrumentation Tool," Intel. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
[20] C.-K. Luk et al., "Pin," ACM SIGPLAN Notices, vol. 40, no. 6. Association for Computing Machinery (ACM), pp. 190–200, Jun. 12, 2005. doi: 10.1145/1064978.1065034. Available: http://dx.doi.org/10.1145/1064978.1065034
[21] "X86 Encoder Decoder: X86 Encoder Decoder User Guide." Available: https://intelxed.github.io/ref-manual/
[22] h3xduck, "GitHub - h3xduck/TaintBlade: A Protocol Reverse Engineering Framework" GitHub. Available: https://github.com/h3xduck/TaintBlade
[23] "UPX: the Ultimate Packer for eXecutables - Homepage." Available: https://upx.github.io/
[24] "Oracle VM VirtualBox." Available: https://www.virtualbox.org/
[25] Mandiant, "GitHub - mandiant/flare-vm: A collection of software installations scripts for Windows systems that allows you to easily setup and maintain a reverse engineering environment on a VM.," GitHub. Available: https://github.com/mandiant/flare-vm
[26] "REMnux: A Linux Toolkit for Malware Analysts." Available: https://remnux.org/
[27] "Overview of Netzob — Netzob 1.0 git documentation." Available: https://netzob.readthedocs.io/en/latest/overview/index.html#step-1-clustering-messages-and-partitioning-in-fields
[28] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher," Proceedings of the 16th ACM conference on Computer and communications security. ACM, Nov. 09, 2009. doi: 10.1145/1653662.1653737. Available: http://dx.doi.org/10.1145/1653662.1653737
[29] "QEMU" Available: https://www.qemu.org/

APPENDIX A - HEURISTIC LEVEL ALGORITHM

Algorithm 1: Algorithm for assigning heuristic level to each byte-level comparison

Input: List of comparison operations from heuristics module
Output: List of keywords, list of delimiters

```
comparison_list ← empty list of byte comparisons
heuristic_level = 0
last_compared_color = 0
for comparison_heuristic in heuristics_vector.comparisons do
    heuristic_colors ← comparison_heuristic.colors
    first_color_in_heuristic ← true
    for ii in 0 to heuristic_colors.size() −1 do
        color ← heuristic_colors[ii]
        if color == last_compared_color +1 then
            last_compared_color ← color
        else if not first_color_in_heuristic then
            last_compared_color ← color
        else
            last_compared_color ← color
            heuristic_level +1
        end if
        comparison_list.append(heuristic_level,
        comparison_heuristic.data)
        first_color_in_heuristic ← false
    end for
end for
```

APPENDIX B - EVALUATION GUI SCREENSHOTS

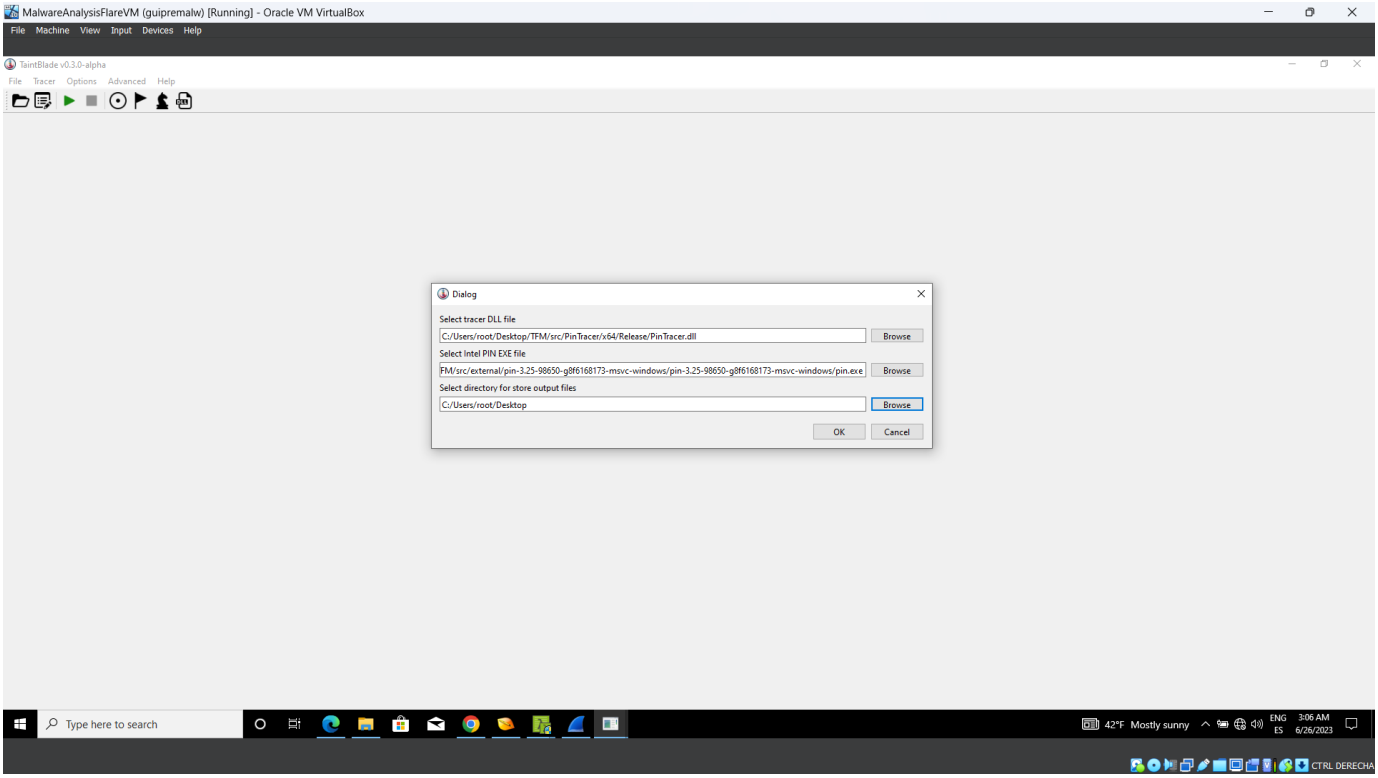

Fig. 9: Selection of TaintBlade options.

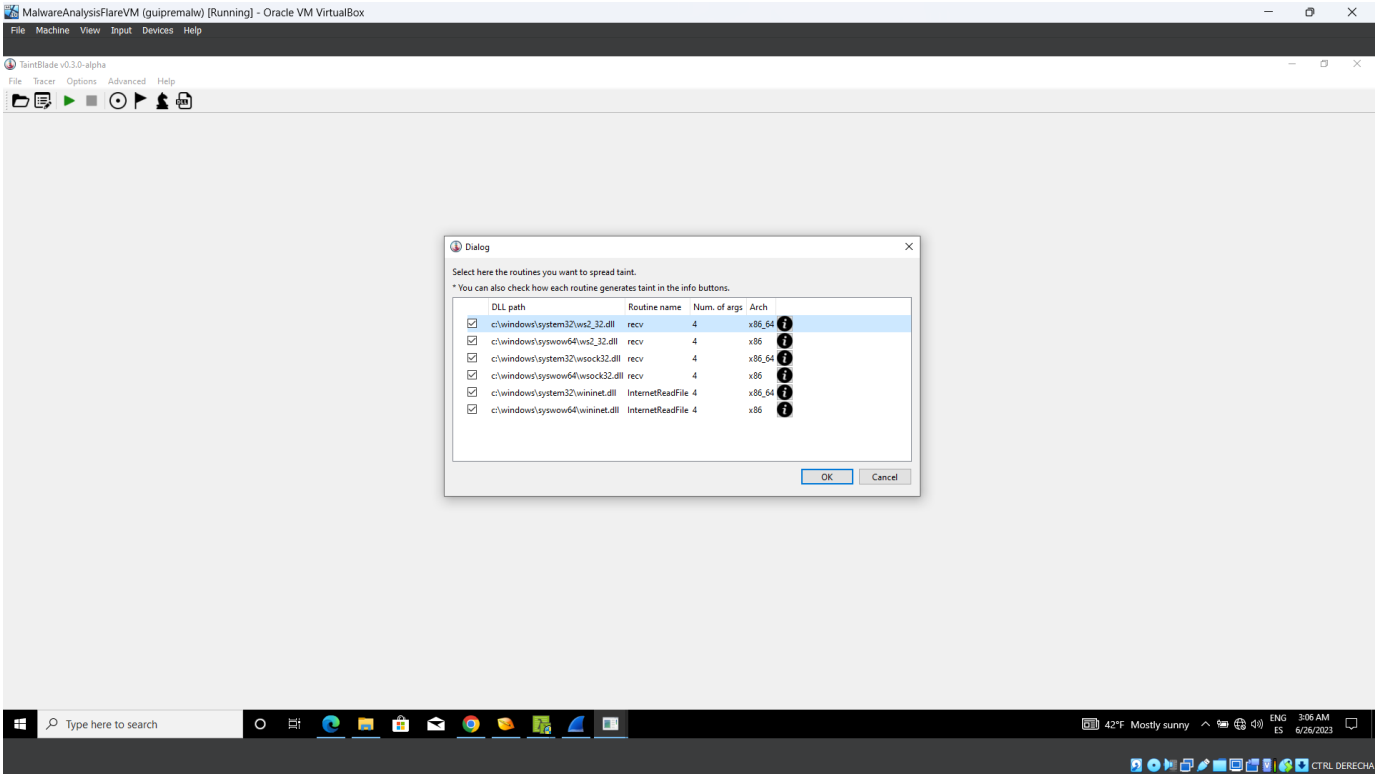

Fig. 10: Selection of taint sources.

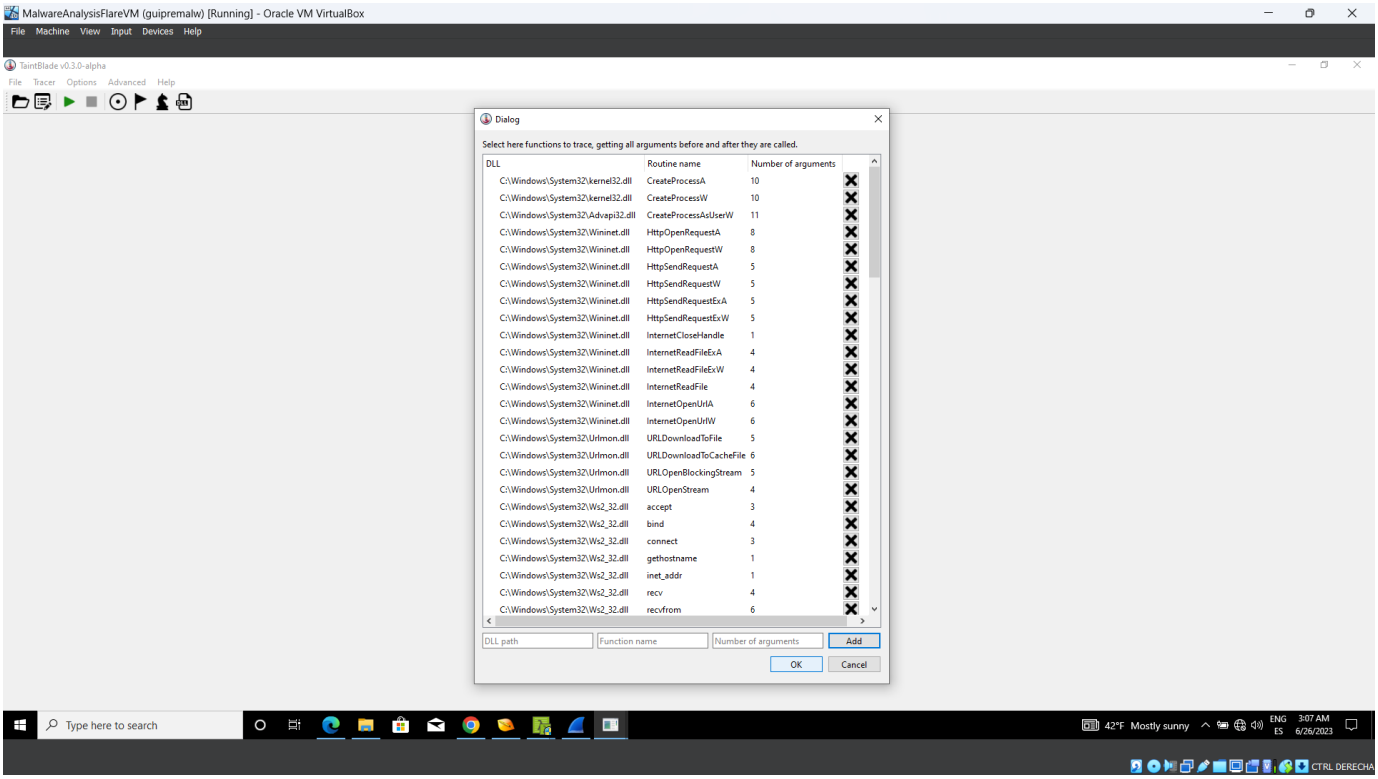

Fig. 11: Selection of trace routines to be instrumented by the tracing module.

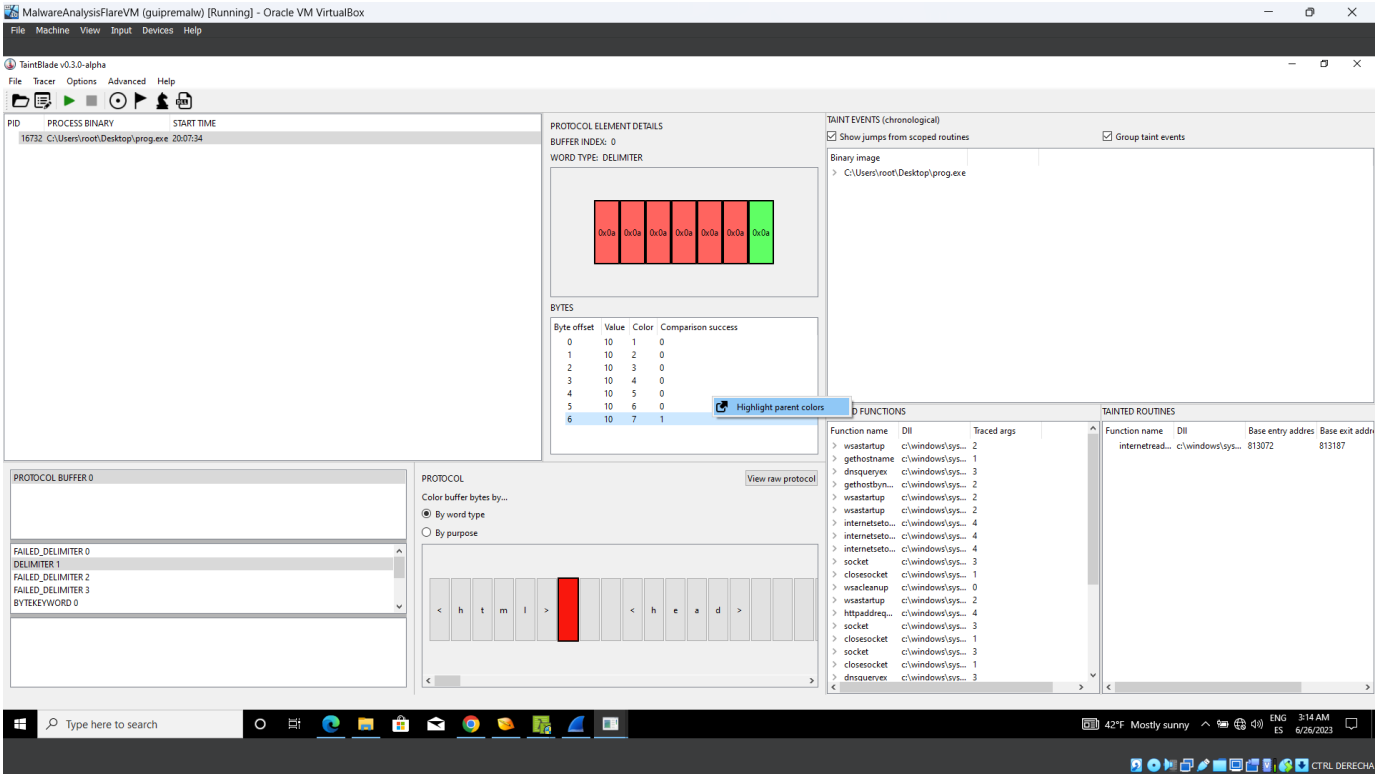

Fig. 12: Results after instrumenting the malware sample with INetSim default configuration.

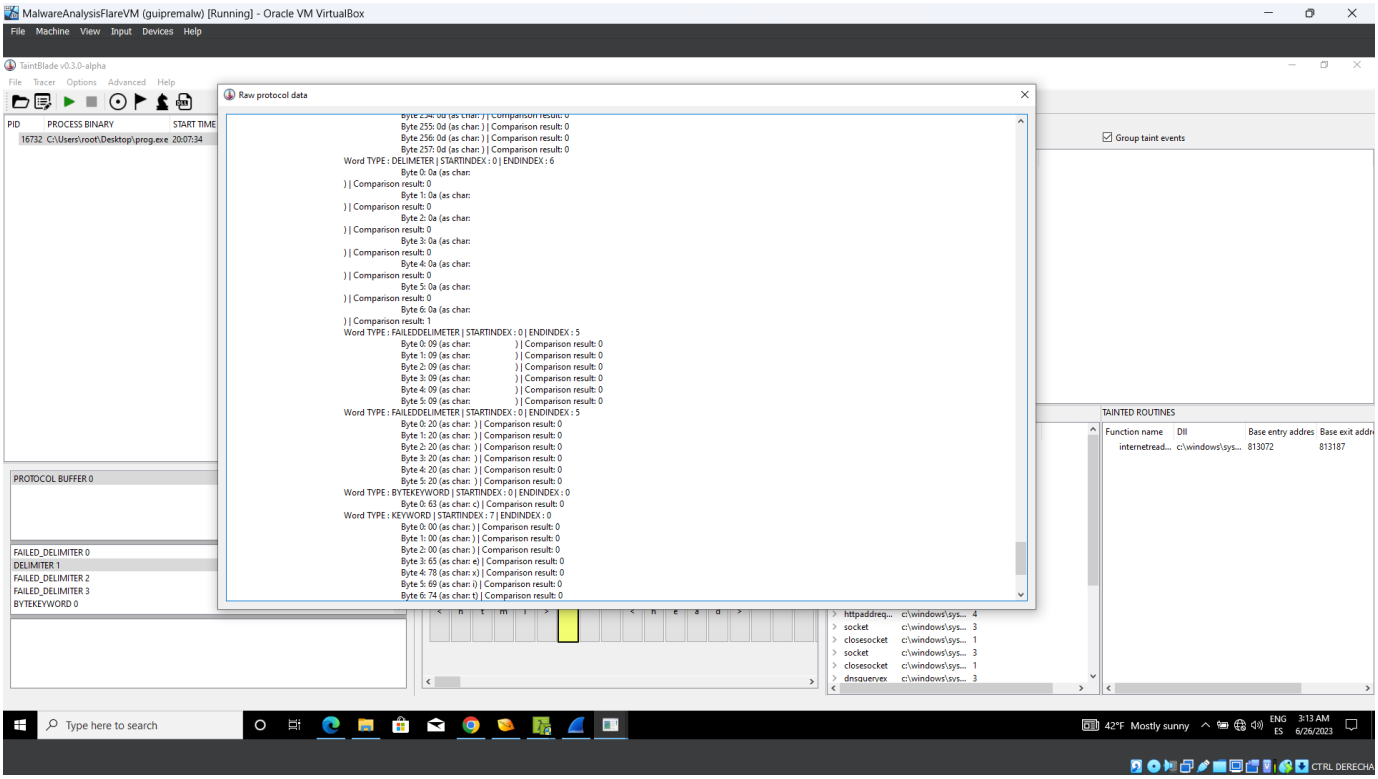

Fig. 13: Raw protocol reversed with INetSim default configuration.

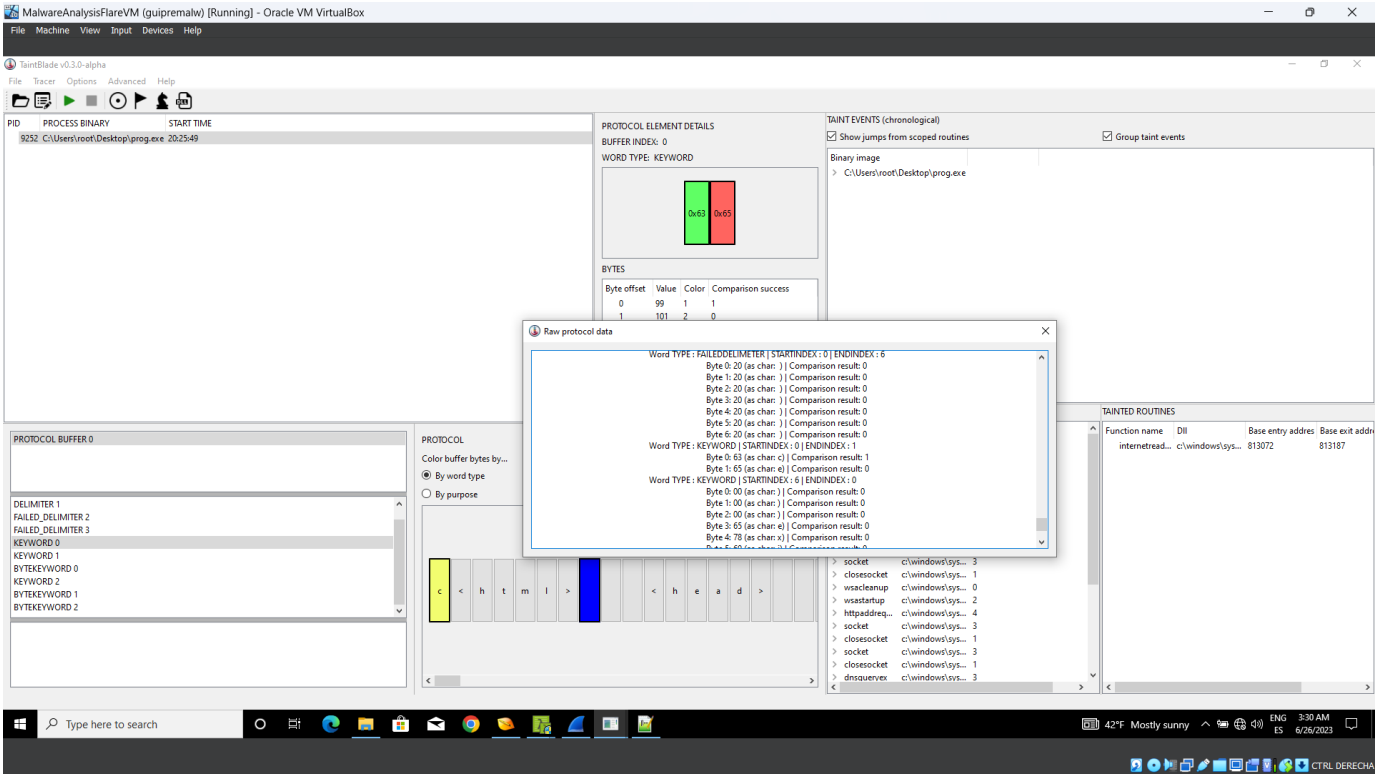

Fig. 14: Raw protocol after receiving a byte array starting with character 'c'.

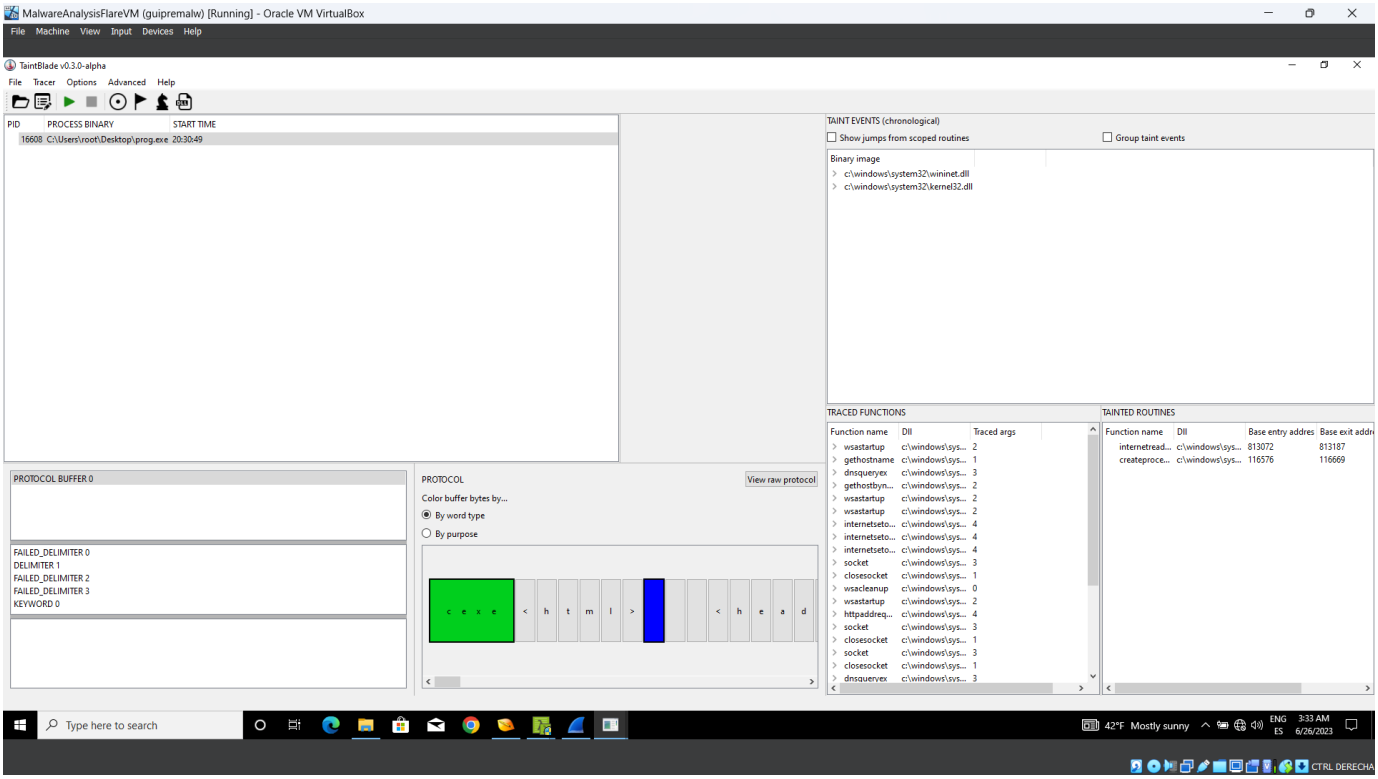

Fig. 15: Results after instrumenting the malware sample with INetSim sending a byte array starting with string 'cexe'.

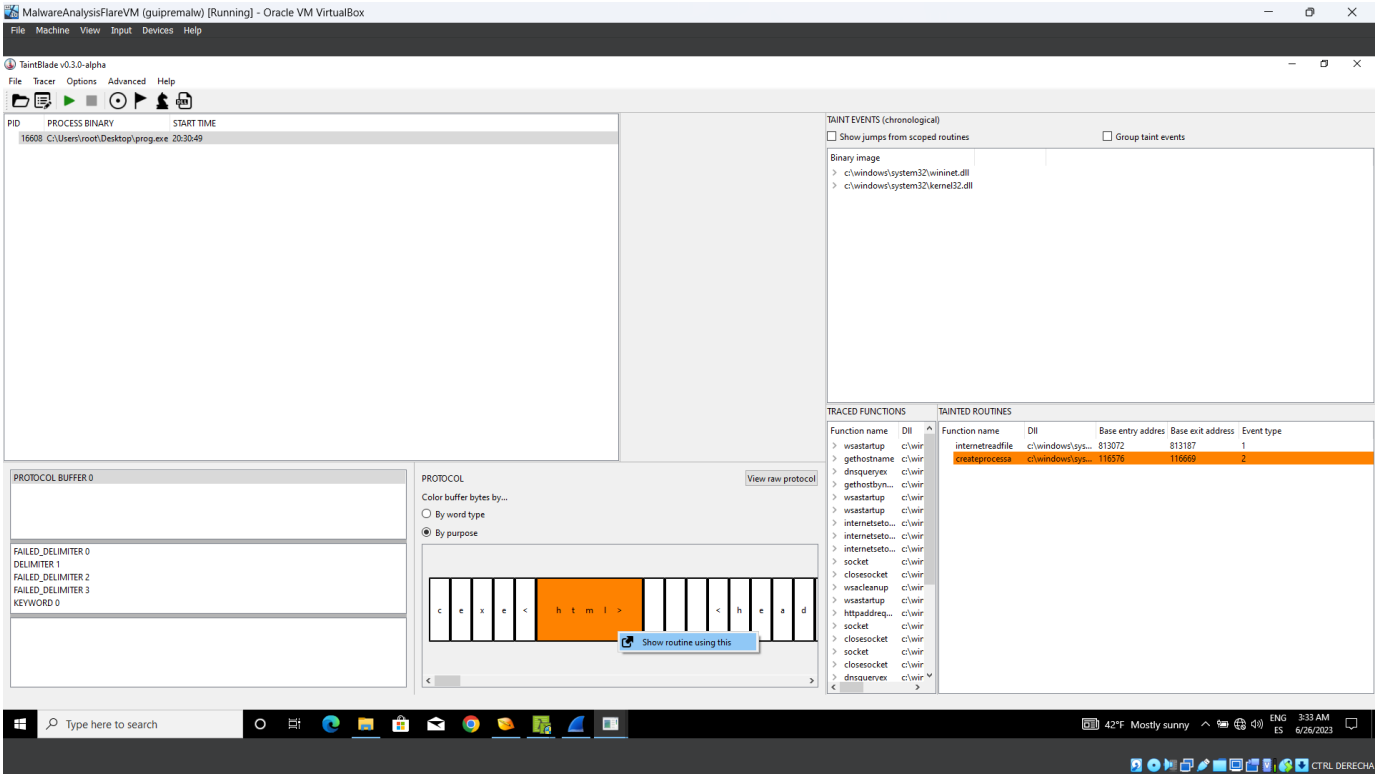

Fig. 16: Semantic information detected by TaintBlade, shows bytes used at a taint sink.

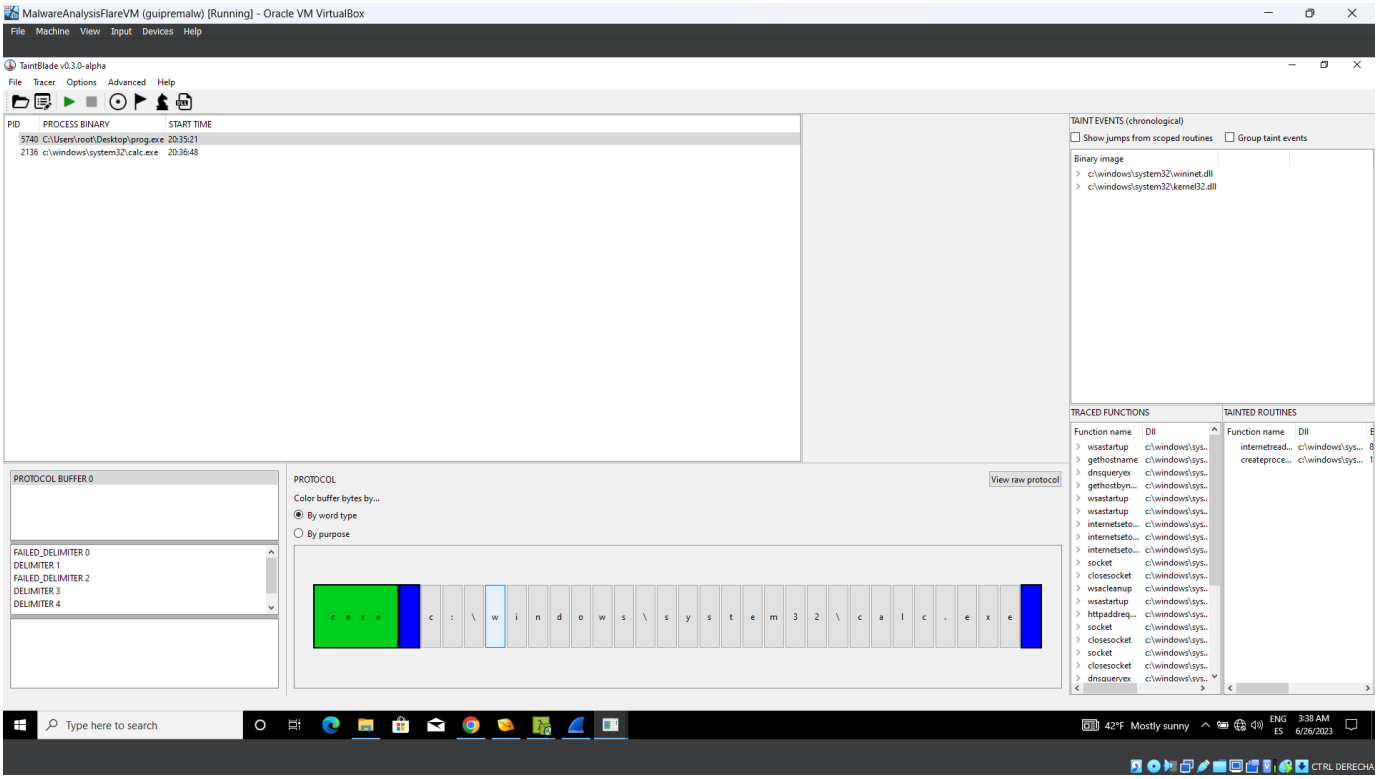

Fig. 17: TaintBlade GUI showing final results for the protocol of receiving a command from the C2.