BLUESPAWN Design and Architecture

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

James McDowell
Spring, 2021

Technical Project Team Members:
Jacob Smith
William Mayes
Calvin Krist

Advisor: Professor Yongwhi Kwon

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature: James McDowell Date: 05/12/2021
James McDowell

Approved: Yonghwi Kwon Date: 05 / 14 / 2021
Yonghwi Kwon, Department of Computer Science

Author's Note: This document is an addendum to *BLUESPAWN: An Open-Source, Active Defense & Endpoint Detection and Response (EDR) Software for Windows-based Systems* (Smith, 2020) intended to focus on the design and implementation of the features described therein. However, as BLUESPAWN is a continuously evolving project, a number of its inner workings have changed and are no longer accurately portrayed in that document. Thus, while the two documents may conflict in some implementation details, those outlined here are the most up to date.

Abstract

BLUESPAWN is an open source endpoint detection and response (EDR) and active defense tool, used to quickly secure a system, identify and remove malicious activity, and continuously monitor a system. The tool was developed in response to a lack of available open source defensive tooling and in an attempt to demystify the blackbox nature of industry EDR products. While existing literature describes the motivations and design philosophy behind and success of BLUESPAWN in much greater detail (Smith, 2020), this paper will focus on BLUESPAWN's architecture and technical design.

Section I - Overview

The BLUESPAWN client currently consists of five primary modules: utilities, hunt, scan, monitor, and mitigate. The utilities module is exactly what it sounds like: it performs the grunt work behind the scenes for the other more developer-friendly modules. The next module is the hunt module, which contains methods for identifying system artifacts that may be used to serve a malicious purpose. However, the hunt module itself doesn't make a determination as to whether

anything is good or bad; that decision is left up to the scan module, which also identifies relationships between detections. The monitor module, as it exists now, automatically detects when changes to the system occur and utilizes the hunt module to determine if they were malicious. The mitigations module is a configurable framework for automatically applying settings and fixing vulnerabilities to secure the system. Each of these modules will be described in more detail below.

Section II - Mitigations

While BLUEPSAWN's primary intent is to serve as an active threat hunting and EDR tool, it also is capable of quickly applying important security settings to a system. This occurs through the mitigation module of bluespawn. Initially, BLUESPAWN sought to map every mitigation applied to a Security Technical Implementation Guide (STIG) released by the DoD (DISA, 2021). Unfortunately, some important security settings weren't included in any STIGs (such as configuring LSASS to run as a Protected Process Light). This, along with the desire to have mitigations be fully configurable, formed the basis for the motivation behind redesigning the manner in which mitigations are organized and applied.

In addition to classifying tactics and techniques, the MITRE ATT&CK framework (MITRE, n.d.) also includes a classification for mitigations. Given the flexibility and extensibility of this framework, particularly as compared to DoD STIGs, the BLUESPAWN mitigation module was built around this framework instead. Under this framework, every setting and change BLUESPAWN's mitigation module makes is placed under one of the MITRE mitigations. For example, configuring LSASS to run as a Protected Process Light falls under M1025, Privileged Process Integrity. Each individual change BLUESPAWN may make as part of

its mitigations is referred to as a mitigation policy. Currently, support exists for four types of mitigation policies: registry value policies, registry subkey policies, event log configuration policies, and combined policies, which enable a mechanism to express basic logic in which changes are needed. Support is also underway for firewall configuration policies.

The ability for users to configure BLUESPAWN's mitigations was at the forefront of the design process. Each mitigation policy includes a default enforcement level, which can be used to avoid enabling mitigations that may come at a cost. For example, disabling wscript can be a valuable mitigation, but in environments where wscript is needed, users may run BLUESPAWN at a lower enforcement level to avoid enforcing that mitigation policy. Alternatively, users may run individual MITRE mitigation categories at a lower enforcement level, or if desired, even a single mitigation policy. This is configurable by a JSON file provided when running mitigations. Additionally, if users desire mitigations not included by default, BLUESPAWN will accept a JSON file describing the mitigation policies to be added and enforce them along with the defaults.

Though significant progress has been made on this module, it is still in its infancy. Future work should consider automatically detecting 3rd party software and services installed on the system and providing options for mitigating them. Future work may also consider implementing functionality similar to that of 0patch (ACROS Security, n.d.), patching vulnerabilities in software installed on the system. Finally, future work may focus on making system permissions and privileges meet industry best practices.

Section III - The Core of BLUESPAWN

The rest of BLUESPAWN is oriented around identifying any artifact on the system that

may be indicative of malicious activity. These artifacts may be as simple as event log entries or more complex, such as an anomalous region of memory in a process. Any such artifact identified as worth further investigation by BLUESPAWN will be referred to as a detection, regardless of whether or not it is eventually determined to be malicious.

The primary mechanism used to identify such detections is known as a hunt. The MITRE ATT&CK Framework (MITRE, n.d.) has done a fantastic job of generalizing and categorizing techniques that an attacker may use, and BLUESPAWN takes advantage of this. Each supported technique in the framework corresponds to a hunt inside BLUESPAWN, which aims to identify all possible system artifacts that could be created when an attacker executes that technique. For example, run keys are registry artifacts that can be leveraged by attackers to achieve persistence, and hunt T1547 enumerates all such run keys. While hunts do seek to identify artifacts worth investigating further, they don't generally determine whether such artifacts are benign or malicious. Rather, they create a Detection object referencing the underlying system artifact. This detection is then passed to the scan module, which is responsible for making such determinations.

This was not the initial design. Instead, each hunt was responsible for doing all of the work on all detections. Back to the example of the T1547 hunt, in the old model, the hunt would have been responsible for finding the registry values, reading their contents, finding the associated file, and determining if the command to be executed would be benign. This is remarkably similar to the hunt that would look through all installed services, which would have been responsible for finding the ImagePath registry values, reading their contents, finding the associated file, and determining if the command to be executed upon service startup would be benign. In fact, many hunts ended up performing the same work since different techniques often

rely upon the same underlying categorization of artifacts. Thus, a separate scanning module was introduced to help streamline this process.

The Scanning Module

The scan module performs a number of functions that are essential for BLUESPAWN's operation. On a high level, three main capabilities are exposed: the ability to perform a "quick" scan of a system artifact to determine whether it is worth further consideration, the ability to perform a full scan of a detection, and the ability to scan the system for artifacts related to an existing detection. While the full scans and association scans are only used internally within the scan module, the quick scans are exposed to other components of BLUESPAWN that may benefit from their presence. Indeed, while hunts are responsible for determining whether or not to create a detection from a system artifact, they often use quick scans to inform this decision.

However, there are some situations where having the scan module as the sole component responsible for determining the maliciousness of a detection doesn't work particularly well. Consider the case of modifications to a Subject Interface Package (SIP)'s configuration. While SIPs are configurable, in practice, changes made to a SIP's configuration are almost always malicious. BLUESPAWN's hunt T1553 is able to check for modifications to SIPs' configurations. If it finds any changes to the registry, the associated registry value is practically guaranteed to serve a malicious purpose, even though it may reference a benign file such as NTDLL. Meanwhile, hunt T1112 could search the registry for any reference to suspicious files, and in the context of hunt T1112, a SIP configuration being changed to reference NTDLL instead of WinTrust.dll wouldn't be malicious at all. The takeaway here is that a detection must be considered in *context* of the hunt responsible for identifying it, as it can indicate that an

otherwise benign-looking artifact may in fact be malicious. Once a detection has been passed to the scan module from a hunt, while the hunt responsible is recorded, integrating that information directly with the scanning process would be challenging. Thus, when hunts register a detection with the scan module, rather than tasking the scan module to figure out the context in which it was identified, the hunt provides a contextual maliciousness score along with the detection.

Scoring Detections with Certainty

This concept of assigning a score to a detection is not unique to the contextual maliciousness; in fact, that is one of the primary end goals of the scan module. As deep scans and association scans are performed, the results are also used to compute the detection's maliciousness score. These scores are intended to represent the probability that a detection is associated with malicious behavior in some way and are therefore referred to as certainties. Each detection has three conceptual types of certainty. The simplest of these is the intrinsic certainty, which is the certainty derived from the artifact itself. For example, if a file is unsigned, it may be assigned a non-zero intrinsic certainty. The associative certainty is derived from the associations the detection in question has with the other detections. Finally, the contextual certainty comes from the context of the hunt responsible for creating the detection. However, this certainty is not stored separately; instead, it is combined with the intrinsic certainty score.

Handling of Detections

Once a hunt determines that a detection should be created for a system artifact, the detection gets passed to the scan module. There, the scan module first checks if it refers to the same underlying system artifact as a pre-existing detection. If the detection was not a duplicate,

it is then queued for in-depth scans. First, the detection undergoes a deep scan, which combines its results with the contextual certainty to initialize the intrinsic certainty. After the deep scan, the scanner is given an opportunity to identify associated detections. Association scans may be very resource intensive, and as a result, in some cases, association scans won't be undertaken unless the intrinsic certainty exceeds some threshold. If the detection was a duplicate, it instead adds the identifying hunt to the context of the existing detection and adds the contextual certainty into the existing detection's intrinsic certainty. If this changes the intrinsic certainty, association scans will be given another opportunity to be run in case the intrinsic certainty wasn't previously high enough to trigger the association scans. The new detection is then discarded, as it is no longer needed. Finally, if the combined value between the intrinsic and associative certainty exceeds some threshold, the detection will be marked as malicious and handled as such.

If this threshold is exceeded, two things happen. First, the detection will get logged to all configured detection log sinks. At BLUESPAWN's current state, this can include being sent to the console, saved to a file, or sent over the network using gRPC, though this is an extensible framework. BLUESPAWN also allows users to specify reactions that should be taken to detections, and if a detection exceeds the threshold, it will be sent to each reaction to deal with. As reactions handle the detection, each reaction is given an opportunity to deal with it in order. However, if the first reaction removes the system artifact, other reactions' behaviors might not work. Consider also the case where BLUESPAWN identifies a log that indicates a file which no longer exists was used in a malicious way. A detection would still be created for this file simply to record it, even though it's not present anymore. To handle both of these cases, BLUESPAWN uses the concept of a "stale" detection, or a detection that references an artifact that is

inconsistent with the current state of the system. If a reaction fully handles a detection, it will mark the detection as stale, and other reactions will no longer attempt to remediate it.

Reacting to Malicious Detections

BLUESPAWN has built in five reactions that can handle a detection. However, before any reaction gets a chance to run, the detection's remediator is first given an opportunity. Detections may refer to something that can't be fixed simply by being removed; instead, it may need special handling. In this case, when the hunt creates the detection, it will also pass in a remediator, which if run, would give the detection the special handling it needs. The remediator may also mark the detection as stale after running, or if the remediator wasn't entirely sufficient on its own, it may not. After the remediator is given an opportunity to run, each enabled reaction will have an opportunity to see if the reaction applies to the detection in question. Note that no reactions apply to stale detections. If so, the reaction is run. BLUESPAWN's five reactions are delete-file, which deletes the file, taking ownership and changing permissions if needed; quarantine-file, which changes permissions on the file to restrict everyone's access; remove-value, which removes a registry key or value; suspend-process, which suspends a process; and carve-memory, which is described in more detail below.

The memory carving reaction is designed to handle situations where malware is running in the memory of an otherwise benign process, but restarting the process isn't a viable option. For example, if a critical process is infected on a server that is required to remain online, the memory carving reaction is designed to remediate it without the need for a full system restart. This works in three phases. The first phase walks the stack of every thread of the process in question. If any location in the thread's call stack is within the section of memory marked as

malicious, the thread is immediately terminated. This step is particularly effective at shutting down malware running in injected memory such as meterpreter shells or cobaltstrike beacons. Once the thread is gone, since the memory isn't associated with anything else in the process, no other thread ever enters that memory. However, for malware that is loaded via more legitimate methods (such as a malicious LSA security provider), other threads may later enter that memory region (such as every time a password changes). Since this interaction with the malicious memory may be particularly short lived, simply scanning active threads for those executing code from the malicious memory has a low chance of finding anything. Thus, phase 2 was introduced to search all memory in the process and all registers of all threads for any pointers to the malicious memory. Pointers to non-executable parts of the malicious memory have the data pointed to set to all zeroes, and pointers to executable memory have the data pointed to patched to set rax/eax to zero then return. In both the x86 cdecl and x64 calling conventions, this will cause the function to immediately return upon being called without error, though it should be noted that this fails to properly clean up the stack in x86 stdcall functions. However, given that most malicious memory will be user developed and most user developed code uses the cdecl calling convention by default, this was determined to be an uncommon issue. However, even this may not be enough; if a new library is loaded later, the DllMain function of the malicious memory may be called, and its exports may be loaded and used. Thus, the DllMain function and all exported functions are also patched with instructions to immediately return zero. While effective, this method of carving memory is imperfect. Future work may seek to emulate Process Hacker's functionality allowing it to properly unload a dll, compute the number of bytes required to be cleaned up by stdcall functions, and walk threads' call stack back to outside the malicious memory region rather than outright killing them.

Scanning has much room to grow in terms of future work. As of right now, identifying links between detections is handled pretty simply, for the most part looking for strings that refer to other files or registry keys or looking at the DLLs loaded into processes. However with the wealth of information that tools such as Sysmon can log (Russinovich & Garnier, 2021), BLUESPAWN would be able to much more efficiently identify which process is responsible for creating certain files or registry values and then identify other actions taken by that process. Future work should strongly consider integrating logging into association detection. Additionally, deep scans currently rely on a file's signature and the results of a scan with Yara rules (VirusTotal, n.d.), but machine learning models of malware detection are quickly becoming much more effective. Future work should also strongly consider implementing such a machine learning model or otherwise focus on improvements to the quality of deep scans.

Section IV: Hunting and Monitoring

The mechanism behind hunts was briefly touched upon in sections II and III. In short, a hunt is a short bit of code that enumerates possible system artifacts that may either be malicious or serve as evidence for malicious behavior falling under a specific MITRE ATT&CK technique. As described previously, the scan module is primarily responsible for making a maliciousness determination, but the hunt does have some degree of leeway with the contextual certainty, and the hunt is also responsible for deciding whether to create a detection in the first place.

Each hunt is split up by subtechniques; for example the hunt for T1546 - Event Triggered Persistence is split up into a number of subtechnique hunts, such as Subtechnique 002 - Malicious Screensaver or Subtechnique 007 - Malicious Netsh Helper. Subtechnique hunts are even further divided into subsections, though this is not a part of the MITRE ATT&CK

framework. Instead, the subsections divide the hunt into the lowest granularity possible, often checking only one system artifact per subsection. For example within hunt T1546.002, one subsection checks for a registry key setting the screensaver while another checks the default screensaver file. Other subsections may check multiple artifacts of the same type; for example, any program with a debugger set in its image file execution options will be automatically flagged under the same subsection.

A couple of hunts are of particular note, going beyond basic scans for predefined files, registry values, or logs. The hunt for process injection (T1055) uses a version of Hasherezade's PE Sieve (Hasherezade, 2018) modified to statically link with BLUESPAWN. PE Sieve is an open source tool designed to scan memory for shellcode or improperly mapped images, boasting support for catching process hollowing, process doppelganging, process herpaderping, and more. Integrating PE Sieve into BLUESPAWN's hunting capabilities provides a quick way to very effectively catch a wide variety of in-memory attacks. However, PE Sieve is not perfect; due to complications with the in memory handling of .NET, it is unable to detect when a .NET image has been modified, and at times, it fails to catch in memory shellcode (Orr 2020). Future work should investigate solutions to these issues.

Additionally, given how heavily BLUESPAWN relies on Windows signatures, the hunt for subverting Windows trust controls (T1553) is also of particular note. SpecterOps' paper (Graeber, 2018) on subverting trust in Windows details the ways in which attackers may modify the registry to cause `WinVerifyTrust` to report certain unsigned files or files with invalid signatures are in fact valid. The paper concludes with a table containing the known-good data that should be in the relevant registry keys. BLUESPAWN incorporates this information by compressing it into a resource included in compilation and verifying the system registry matches

it. This prevents any sort of attack that relies on modifying the registry data to redirect signature verification to malicious DLLs instead. BLUESPAWN also ensures that the search path for the relevant DLLs are not being hijacked and that all DLLs involved in the signature verification process are signed themselves. Without this, an attacker might place a DLL with the same name as one used in the signature verification process in the same folder as BLUESPAWN, causing BLUESPAWN to use it instead of the correct DLL. Ensuring that all DLLs are signed is meant as more of a redundancy; however, if a new SIP guid is added, this is intended to ensure that it is not hijacked easily. The final concern lies in trusting the calls made to the APIs that are used to read the registry and verify signatures, which is addressed using PE Sieve. While not infallible, these measures certainly raise the bar for attackers.

Hunts and monitoring go hand in hand. Each hunt is also responsible for subscribing to events that will be triggered when system information relevant to the hunt is changed. This may occur, for example, when a file is modified, a registry value is created, or the Windows event logs record a certain event. When a hunt is run, it also takes in a scope object, which may be used to indicate that only certain parts of the hunt may be run. The scope object specifies which subtechniques and subsections should be run by the hunt. With each event to which the hunt subscribes, the hunt also creates an associated scope. Thus, when the event is triggered, the hunt is rerun with the associated scope, running only the relevant subsections and/or subtechnique hunts.

In BLUESPAWN's current state, monitoring is by far the most underutilized and underdeveloped component. It is limited to the scope of hunts, and while this is much more efficient than naively rerunning the entire hunt, for events that watch entire directories or registry keys, rescanning every file or registry value when any single one of them changes is quite

inefficient. Future work should consider expanding monitoring to include API monitoring, setting up an AMSI interface, and making event monitoring more efficient in handling triggered events.

Section V - Work In Progress Features

Agent7 Integration

Agent7 is an open source security monitoring tool for Windows endpoints aimed at gathering and aggregating security-relation information about systems (Marshall, 2021). This information is all passed to a dedicated Agent7 server that aggregates and displays all of the data, focused on identifying security misconfiguration issues such as insecure file permissions or improperly configured user privileges. While Agent7 and BLUESPAWN sought the same end goal, their capabilities were almost entirely mutually exclusive. Agent7 could aggregate data and take actions across a network at large, but it had very limited threat hunting capabilities. Meanwhile, BLUESPAWN was focused heavily on threat hunting, but its general information gathering was almost nonexistent, and it didn't have any server component at all. With their integration, each system would gain the benefits of the other.

However, Agent7 was written in Python whereas BLUESPAWN was written in C++. To address this, a Python binding was added to BLUESPAWN, which would allow any Python developer to import BLUESPAWN as a Python module and interact with it directly. Agent7 then had BLUESPAWN added to its dashboard and automatically deploys it when installed. This integration allows users to deploy and use BLUESPAWN across an entire network all at once. Note, however, that this integration is still in the alpha phase of development and does not support all of BLUESPAWN's features yet. Beyond that, with Python already partly integrated

with BLUESPAWN, this opens the door for future work to integrate FireEye's CAPA (Ballenthin & Raabe, 2020) or Florian Roth's Sigma (Roth, 2021) for much more powerful scanning and hunting.

Agent DLL

Work has been underway for quite a while to improve BLUESPAWN's monitoring capabilities. One line of effort in this is the creation of an agent dll that gets loaded into every running process. Under the current design, the dll would be responsible for monitoring the APIs used by the process, identifying and stopping malicious calls to Windows APIs, though future work may consider additional uses such as information gathering or more detailed memory scanning.

Many modern EDR products use similar approaches to monitor user-mode APIs to great effect (Eidelberg, 2021). This allows them to intercept calls before they reach the kernel and either change the call to prevent a restricted action, block the call entirely and report failure, or simply record the call. For sensitive APIs such as `NtCreateUserThread` or `NtReadVirtualMemory` that allow interprocess access, being able to block calls to these functions is paramount; otherwise, attackers would be able to do things such as migrate between processes or steal secrets from other processes' memory while evading detection. Microsoft has added support for ETW tracing, but unfortunately, an EDR product relying on ETW tracing can only find out about such a prohibited action after it has taken place and is too late to stop.

BLUESPAWN's implementation of API hooking is still a work in progress. As of now, APIs are hooked using Microsoft's Detours (Microsoft, 2018). Under the current design model, the main BLUESPAWN executable will be responsible for deciding how to respond to calls to

certain sensitive APIs. While the agent may be capable of making such a decision with less of a performance hit, making the decision in the BLUESPAWN executable allows for more tightly integrated decision making. For example, if BLUESPAWN wishes to ensure that certain DLLs with an associated detection identified as malicious are never loaded and the agent alone is responsible for deciding whether to allow the call, it would likely not have that information readily available, and distributing that information to upwards of a hundred processes with each new detection comes with its own performance hits.

Since the main BLUESPAWN executable is to be making the decision, it will need information about the call to decide. Without knowing where a thread is being created, what process is being spawned, what memory is being read, or any information like that, BLUESPAWN can't make an informed decision as to whether or not the associated call should be permitted. Since most raw values passed as arguments such as handles or pointers only make sense in the context of the originating process, the agent needs a mechanism to serialize this data. Additionally, with each intercepted call, a full call stack is generated to pass to BLUESPAWN. This call stack can be used to identify common denominators in suspicious calls; for example, if the same non-windows DLL is involved in multiple calls to blocked functions, there's a reasonable chance that it's malicious.

While hooking and serialization capabilities have been developed, integration with the BLUESPAWN executable is still underway, and only a few hooks have actually been implemented. Security researchers have documented the APIs hooked by popular EDR products (Yiu, 2021), and this now publicly accessible information will be used as a starting point for preparing BLUESPAWN's hooks. Future work will focus heavily on continued development of the agents to improve BLUESPAWN's monitoring capabilities. However, PE Sieve as discussed

earlier does not currently have the capability to exclude known hooks from its scans (Hasherezade, 2018), so future work will also have to consider contributing to PE sieve to support this.

Section VI - Conclusion and Future Work

BLUESPAWN has grown much since its beginning, but it still has a very long way to go. Its design gives it the ability to perform a fairly strong point in time system scan as well as a robust framework upon which it can grow. However, its real time monitoring capabilities and framework still leave much to be desired. While nearly every aspect of BLUESPAWN has a number of avenues for future improvement and growth, future work should focus heavily on building out real time monitoring capabilities.

References

ACROS Security. (n.d.). 0patch. Retrieved from https://0patch.com/

Ballenthin, W., & Raabe, M. (2020, July 16). Capa: Automatically Identify Malware Capabilities. Retrieved from https://www.fireeye.com/blog/threat-research/2020/07/capa-automatically-identify-malware-capabilities.html

Defense Information Systems Agency. (2021, May 11). Security Technical Implementation Guides (STIGs). Retrieved from https://public.cyber.mil/stigs/

Eidelberg, M. (2021, February 2). Endpoint Detection and Response: How Hackers Have Evolved. Retrieved from https://www.optiv.com/insights/source-zero/blog/endpoint-detection-and-response-how-hackers-have-evolved

Graeber, M. (2018) *Subverting Trust in Windows* [Whitepaper]. SpecterOps. https://www.specterops.io/assets/resources/SpecterOps_Subverting_Trust_in_Windows.pdf

Hasherezade. (2018, November 27). PE-sieve. Retrieved from https://hshrzd.wordpress.com/pe-sieve/

Hasherezade. (2018, February). Whitelisting known hooks. Retrieved from https://github.com/hasherezade/pe-sieve/issues/4

Marshall, B. (2021). Home | Agent7. Retrieved from https://agent7.sec-eng.tech/

Microsoft. (2019, August 14). Detours. Retrieved from https://www.microsoft.com/en-us/research/project/detours/

MITRE. (n.d.). MITRE ATT&CK®. Retrieved from https://attack.mitre.org/

Orr, F. (2020, August 03). Masking Malicious Memory Artifacts – Part III: Bypassing Defensive Scanners. Retrieved from https://www.cyberark.com/resources/threat-research-blog/masking-malicious-memory-artifacts-part-iii-bypassing-defensive-scanners

Roth, F. (2021). Sigma. Retrieved from https://github.com/SigmaHQ/sigma

Russinovich, M., & Garnier, T. (2021, April 21). Sysmon - Windows Sysinternals. Retrieved from https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon

Smith, J. (2020). *BLUESPAWN: An Open-Source, Active Defense & Endpoint Detection and Response (EDR) Software for Windows-based Systems* (Undergraduate thesis). University of Virginia. doi:10.18130/v3-b1n6-ef83

VirusTotal. (n.d.). YARA - The pattern matching swiss knife for malware researchers. Retrieved from http://virustotal.github.io/yara/

Yiu, V. (2021, May 10). EDRs. Retrieved from https://github.com/Mr-Un1k0d3r/EDRs/blob/main/EDRs.md