{deepfactor}

WHITEPAPER

Observing Application Behavior via API Interception

// Contents

// Challenges to Understanding Application Behavior

Given the rapid growth and evolution of cloud native applications, there are many reasons for engineering teams to observe application behavior during every stage of the software development lifecycle (SDLC). From guaranteeing the workload is performant and stable, to certifying functionality and security hardening, today's developers require robust runtime analysis to ensure applications are behaving as designed. With careful investigation of the application's behavior, engineering teams can confidently—and quickly—move from development, through testing, into production.

This requirement becomes especially important as today's software developers increasingly use third-party software imported from the Internet to assemble applications. According to Gartner, "open-source software is used within mission-critical IT workloads by more than 95% of IT organizations worldwide."[1] For many reasons—speed and flexibility, most notably—software development has evolved well-beyond the contributions of a single developer writing code from scratch!

Though the practice of assembling applications from existing libraries and joining them together with custom code is nearly commonplace, it's not entirely risk free. For example, in 2021, software supply chain attacks grew by more than 300%[2] , indicating the level of security across software development environments remains low. This is one of many examples to highlight how developers have struggled to thoroughly understand and assess the expected behavior of constituent modules being imported.

With these challenges in mind, what exactly can engineering teams do to ensure the overall integrity of the applications they're developing?

In 2021, software supply chain attacks grew by more than

300%

// Introduction to API Interception

There are several solutions available to ascertain the behavior of a given application. For example:

- Static Application Security Testing (SAST) and Software Composition Analysis (SCA) are focused on statically scanning the application's makeup (also known as "Bill of Materials") to identify vulnerable components.
- Application Performance Monitoring is the practice of tracking key software application execution metrics to ensure system availability, optimize service performance, and improve response times.
- Cloud Management platforms enable organizations to manage and optimize cloud (i.e. on-prem and/or private) services and resources, such as provisioning and managing the lifecycle of the cloud infrastructure being used by the application.

1 Gartner, **"A CTO's Guide to Top Practices for Open-Source Software"**, 21 March 2021, Arun Chandrasekaran, Mark Driver. GARTNER is a registered trademark and service mark of Gartner, Inc. and/or its affiliates in the U.S. and internationally and are used herein with permission

2 VentureBeat, **"Report: Software supply chain attacks increased 300% in 2021"**, 27 January 2022, VB Staff.

> However, for developers hoping to specifically understand application behavior, particularly the millions of interactions between imported and custom code, it's important to observe the application in runtime, before shipping into production.

API Interception is the practice of "hooking" and redirecting calls to an application programming interface (API) into custom code. Though API interception can be used to alter application functionality, developers can also use the technique to observe application behavior in near real-time. By observing which APIs are called, what parameters are passed to those APIs, what return values are handed back to the application, how long the APIs took to execute, and so forth, API interception can help the developer understand end-to-end application behavior.

There are many ways to perform API interception, and the objective of this whitepaper is to outline the most common techniques, and detail strengths and weaknesses of each approach. In addition, the last section of this whitepaper reviews how Deepfactor, a developer security platform, has used API interception for the purposes of providing engineering teams with contextual application security insights.

Which APIs Are Important To Intercept?

Before exploring the specifics of various interception techniques, it's prudent to describe the types of APIs generally used by applications. This information can be helpful to determine which APIs to analyze to best understand various application behavior.

That said, from the lowest architectural layer to the highest:

Type of API	Description
System Calls	Programmatic way in which applications request a service from the kernel of the operating system it is executed on. Typically, an operating system will have a few hundred system calls spanning memory management, file operations, network connections, and so on.
Base Library Calls	Simple runtime library (e.g. libc) for modern programming languages found on Linux and UNIX operating systems. Provides additional parameters —such as checking and/or error handling—for system calls, and functions such as random number generators, string manipulation, etc.
Higher Level Library Calls	Delegates to lower base libraries. There are thousands of libraries in this class, including glib, openssl, protobuf, and so on.
Web APIs	Framework that helps create and develop RESTFUL services; transactions occur over the web using the HTTP protocol. Examples include payment processing APIs, CSP (cloud service provider) APIs, or any number of REST APIs exposed by network-facing services.

// **Table 1: Description of the types of APIs used by applications.**

Given the category descriptions above, applications—and the underlying operating system on which they run—expose many viable APIs for interception. However, intercepting the "incorrect" set of APIs can waste valuable time, leaving engineering teams with less opportunity to triage issues, improve stability and performance, and implement new features. Furthermore, identifying the wrong APIs can jeopardize the security of the application, possibly leading developers into a false sense of confidence.

Obviously, this presents a challenge—how do developers even begin to answer the question of which APIs to intercept? In many cases, this decision can be driven by experience and personal bias.

For example, web developers who work exclusively with RESTful applications, may conclude that web APIs are the only APIs worth intercepting to fully understand application behavior, security, and performance. However, assuming that a developer exclusively used a tool or platform for intercepting and analyzing said web APIs, they might miss identifying vulnerable libraries, such as OpenSSL, being used by the web framework. Regardless of the security of the web APIs, the underlying operating system and application would remain exploitable.

The same could be said for exclusively intercepting low-level APIs. Developers or operation teams who chose to narrowly observe the stream of system calls being invoked by the application could fail to notice contextual security clues only evident in higher level library calls. As a result, developers would be unaware of a SQL injection vulnerability or that the encryption library is using deterministic random numbers.

So, assuming there's a chance of omitting valuable information through the interception of a single layer of APIs, engineering teams should then ensure complete coverage by intercepting all APIs, right? While technically true, this is obviously an impossible task! This especially true when you consider many developers are unable to produce an exhaustive list of APIs used in a given application.

Rather, in order to answer the question, "What APIs should we observe (i.e. intercept) to understand our application's behavior?" engineering teams should strive to understand the answers to the following questions:

WHAT IS THE PURPOSE BEHIND INTERCEPTING AND ANALYZING THE APIs?

i.e. what is the overall objective/outcome for inspecting application behavior?

IN WHAT STAGE OF THE APPLICATION'S LIFECYCLE DOES THE INTERCEPTION NEED TO HAPPEN?

i.e. will you be observing the application during development, testing, or production?

WHAT TYPE OF API NEEDS TO BE INTERCEPTED?

i.e. what technological considerations are needed to successfully capture the right information?

By thoroughly answering these questions, engineering teams can begin to create well-informed strategies around the appropriate API intercept technique. Ultimately, the goal is to obtain a behavioral fingerprint of the application by using API interception to create a list of used—and not used—APIs. This fingerprint can be used by developers as a baseline for later comparison, such as performing drift analysis between releases, and prioritization and remediation of open vulnerabilities.

// API Interception Techniques

Note: The following analysis is focused on library and system call APIs. Web APIs require a different solution and will be reviewed later in this whitepaper.

There are several common technological requirements to consider when evaluating the various options available for API interception. In addition to identifying when and where a desired API is called, the API interception technique must never:

MODIFY THE APPLICATION'S BEHAVIOR

i.e. the technology should be seemingly "invisible"

IMPACT THE APPLICATION'S PERFORMANCE AND STABILITY

i.e. the technology should be highly performant and technically sound

INTRODUCE ADDITIONAL SECURITY RISKS

i.e. the technology should not require the application to run in a way that exposes additional risk

However, regardless of the technique used to observe and log application behavior, control must be diverted to the interception tool at the exact moment the API is called to conclusively identify usage and purpose. Considering the requirements outlined above, this can be achieved several different ways:

Instrumentation Technique	Description
Inline Hooking	Rewrites the first part of a monitored function to divert control flow to the API interception tool which then logs any desired information about the API being called.
Callbacks	Requests that the operating system perform the API monitoring and callback to the API interception tool (or run some code provided by the tool) when the selected APIs are called.
Import Table Rewriting	Requests the operating system call the interception tool's own library functions in preference to those of existing libraries. The API interception tool receives the invocations first, then logs any desired information about the API being called.
Method Rewriting	Appends callback code at the start of each function/method being loaded by the application to log desired information about the API being called.

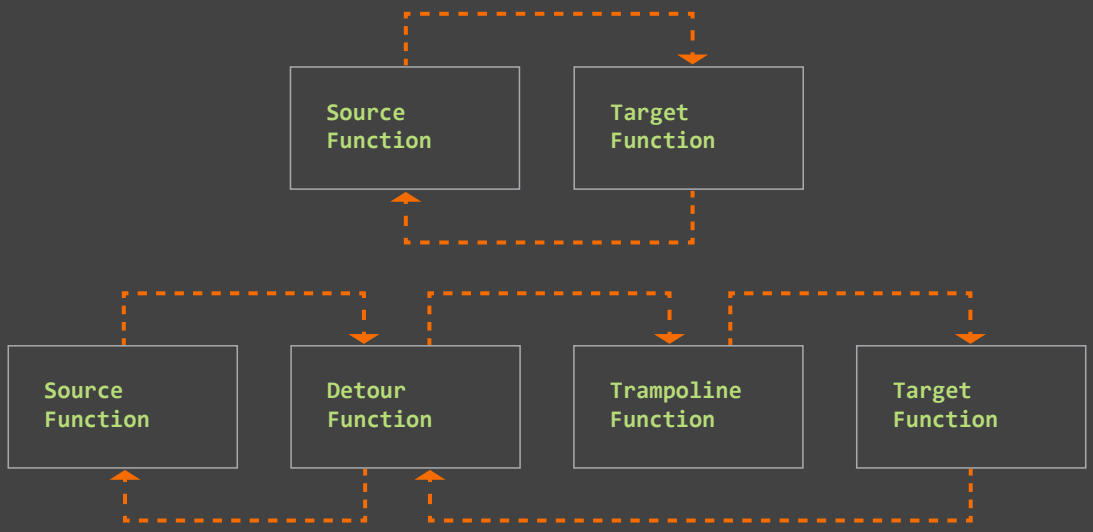
// **Table 2: Description of various API interception techniques**

Inline Hooking

Inline hooking is a technique used to reroute function invocations to a different location (generally) during the launch of a program. This is accomplished by overwriting the first few bytes/instructions in a function with a short, or far, jump to some other location for processing, such as a function inside the API interception tool. Regardless of the location, the jump will destroy at least one of the original instructions in the function. Examples of inline hooking include Detours for **Windows** and **Linux**.

In the case of using inline hooking for API interception, control flow is diverted to the API interception tool, where the program must identify which function was called—this is because more than one function might have been hooked. This is accomplished by a) diverting control flow to a unique place in the API interception tool for each hooked function, or by b) diverting control flow to a common location and walking the stack to determine which function was previously called.

The following diagram, Figure 1, compares the process of normal execution and the control flow changes after inline hooking has occured:



// **Figure 1:**
Application execution with and without inline hooking.

Once the API interception tool logs the necessary information about the invocation, control flow must return to the original function just after the initial hook occurred. However, before this can happen, the instructions destroyed by the original hook need to be replayed.

Depending on the operating system and API, overwritten instruction(s) may take many forms, necessitating the implementation of a full or nearly-full instruction emulator inside the inline hooking module. This complicates the implementation of the solution and can lead to fragility of the module when it's not kept up to date. This includes maintaining support with new CPU instruction sets and ensuring the overwritten instructions remain supported/recognized by the hooking module.

There are several positive aspects to inline hooking:

> The overhead to detour the invocation to the API interception module is minimal, with just a single instruction (plus the decoding required to emulate the overwritten instructions) required.

> In addition, inline hooking can be used in scenarios where the application is statically linked, which is not necessarily true of other interception techniques, such as import table rewriting.

However, inline hooking can have a number of limitations:

> While stacking/chaining is supported, there are limitations when multiple inline hooking systems are used in conjunction. Notably, extra care must be taken to ensure each hooking module is capable of replaying the overwritten instructions from the preceding module in the chain.

> Hooks can be extremely difficult to [safely] remove from running applications. Since most hooks are placed before the application starts, they are not subject to partially patched conditions in the application. However, the same is not true when reversing the process —only one hook can be removed at a time, resulting in a situation where the application is always partially patched. This restriction also applies to stacking hooks, as described previously.

> Processor architectures without coherent instruction/data caches make hooking after the application challenging. This is because placing a new instruction into the instruction stream (via data write), where a different CPU is executing instructions fetched from the same instruction cache line, might end up executing the previous, unpatched instructions. Inline hooking modules must be carefully written to avoid cases like these.

Callbacks

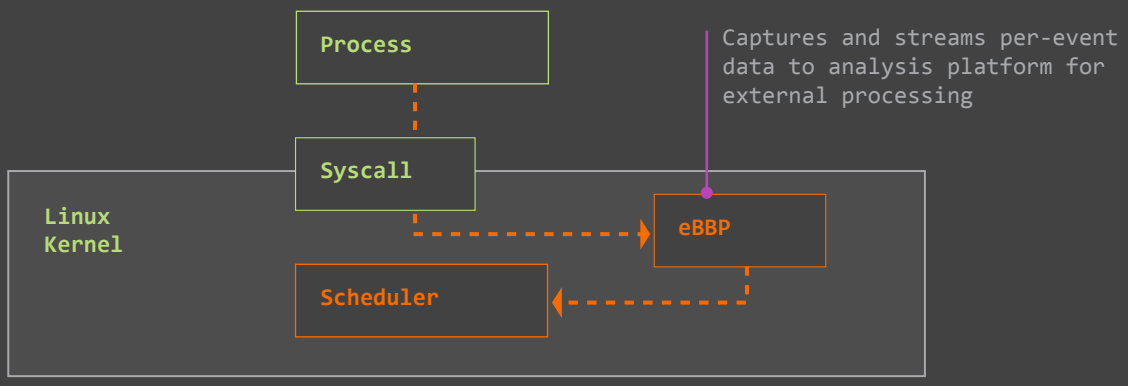
Whereas inline hooking relies on the CPU to communicate with the API interception tool, some operating systems provide a callback interface to register "interest" in a particular set of application APIs. Callbacks enable the operating system to execute a code snippet—generally provided by the API interception tool—at the moment any API of interest is called.

The most popular type of callback is **Extended Berkeley Packet Filter (eBPF)**. With eBPF, the API interception tool can attach small code fragments, also known as programs or snippets, to desired system call(s). eBPF programs are written in the BPF language, and are event-driven, which means they run whenever the kernel or application passes a certain hook point. This can include system calls, function entry/exit, kernel tracepoints, network events, etc.

There are various ways to install and execute eBPF programs/code snippets:

1. **Sidecar:** Container that runs alongside the application container for the purpose of externally monitoring application behavior.
2. **Software Agent:** Daemon program that is installed on the host/node running the application which needs to be monitored.
3. **prctl System Call:** Monitoring program that bundles eBPF code to be dynamically loaded directly into the kernel at startup.

The difference between sidecars and software agents is mostly packaging, and the requisite steps used to provision the system under observation. On the other hand, using the prctl system call generally requires source code modification. However, regardless of the eBPF implementation used, the kernel will execute BPF code snippets—which have been configured to observe specific application APIs—to record and stream system call information to the interception tool. This is pictured below:



// **Figure 2:**
Simplified overview of eBPF architecture and process.

There are several benefits to using callbacks, such as eBPF. Most notably, eBPF offers a robust, feature-rich environment for programmability (although, this is not without its drawbacks—more on that below), making its unified framework convenient to operationalize. In addition, because eBPF doesn't modify application code, there is zero-to-minimal impact to application behavior.

Note: Regardless of the specific interception technique, there's always a chance of adding performance overhead to critical code paths leading into the interceptor program.

As mentioned above, eBPF does have several limitations:

> eBPF is a recent addition to many operating systems, having been added and refined in Linux over the past several years, and only supporting Windows in 2021. This creates a complex **support matrix** that challenges adoption and limits the number of systems where an eBPF-based API interception tool is supported. For example, according to **RedHat**, eBPF is "provided as Tech Preview and thus doesn't come with full support and is not suitable for deployment in production."

"[eBPF is] provided as Tech Preview and thus doesn't come with full support and is not suitable for deployment in production."

—REDHAT

- eBPF is not without its **growing pains**, having exhibited over **two dozen high-severity CVEs** in the past three years, including LPE-to-root vulnerabilities. This steady flow of new CVEs has led some enterprises and verticals (such as FinTech) to restrict which eBPF tools can be deployed—or to disallow eBPF-based solutions entirely.
- eBPF is limited to intercepting system calls, which means higher-level APIs are not observable. This limits eBPF to streaming low-level system calls back to the developer. That said, recent versions of eBPF do provide uprobes, which can be used to observe certain higher-level APIs. However, the **lack of uniform support** limits the usefulness of this feature.
- eBPF requires specific permissions to be granted, such as allowing the application to run code inside the host kernel. While this may be acceptable for development environments, this is not always possible, either due to company policy or technical limitations such as services running in the public cloud.

Import Table Rewriting

Import table rewriting instructs the operating system to modify the table responsible for describing which external dependencies an application relies upon during execution. Using this modification, tools—such as API interception platforms—can reference any version of any library API that will be used in preference before the native/system version. In addition, this technique can be used to supply system-call wrapping base library calls, exposing additional functionality to the interception tool—more on this below.

Whereas inline hooking requires modification to application code bytes in memory, import table rewriting simply modifies the dynamic loader to intercept and wire up (any) library calls to a different location. This means import table rewriting can be used to capture and observe both base and higher level library calls. Though this technique does not allow for the direct interception of system calls (similar to eBPF), intercepting the base library call wrappers for system calls is typically sufficient.

Import table rewriting is usually implemented in Linux/UNIX systems using the LD_PRELOAD environment variable. Setting LD_PRELOAD instructs the operating system to preload the libraries/functions listed in the variable, thus prepending them into the runtime dynamic linker's (loader) resolution cache. Subsequently, when the application is loaded and has an import dependency on a specific symbol—assuming the symbol exists in the LD_PRELOAD libraries—the loader will perform a redirect to the new location.

// **Figure 3:**
Example of import table rewriting using malloc and LD_PRELOAD.

This is illustrated in Figure 3, where import table rewriting is used to set LD_PRELOAD with a unique malloc function that is being used to perform additional checks, such as size limits, logging the allocation for debugging purposes, etc. And because of LD_PRELOAD, the unique malloc function is called before the system's malloc function. On completion of the library's malloc, it typically will "chain" to the original malloc implementation to perform the real operation.

The process of using import table rewriting with LD_PRELOAD to perform API interception offers several advantages:

- In most circumstances, because it doesn't require instruction emulation (i.e. inline hooking) or use an interpreter (i.e. callbacks), important table rewriting offers the best overall performance*.
- Import table rewriting can intercept APIs not available to eBPF. For example, LD_PRELOAD can be used to intercept the rand API (which is a deterministic random number generator), or the notably unsafe functions operating on unchecked length strings (e.g. strcpy). These APIs never result in system calls, thus they are unable to be observed by callback techniques.

The one notable drawback to using LD_PRELOAD for import table rewriting is the absence of support for statically linked applications, which do not use the dynamic linker (loader). In this scenario, either inline hooks and/or eBPF must be used.

Method Rewriting

Method rewriting is an API interception technology that works with bytecode-interpreted languages, such as Java and .NET framework applications. This approach allows API interception tools to prepend custom code to methods being used by the application.

Similar to inline hooking, the interpreter (e.g. Java Virtual Machine) will execute the additional code first, followed by the original code. Generally speaking, the code added to each method is short, usually focused on logging which method was called, including parameters and timestamp.

Method rewriting is only suitable for programming languages that support a "class load callback," or something equivalent. Because method rewriting actually makes changes to the application code, it shares many of the same disadvantages as inline hooking—extra care must be taken by the API interception tool to avoid altering the application's behavior, and there's no technical way to prevent subsequent method rewrites from removing, or otherwise damaging, a previous code addition made by a different tool.

*

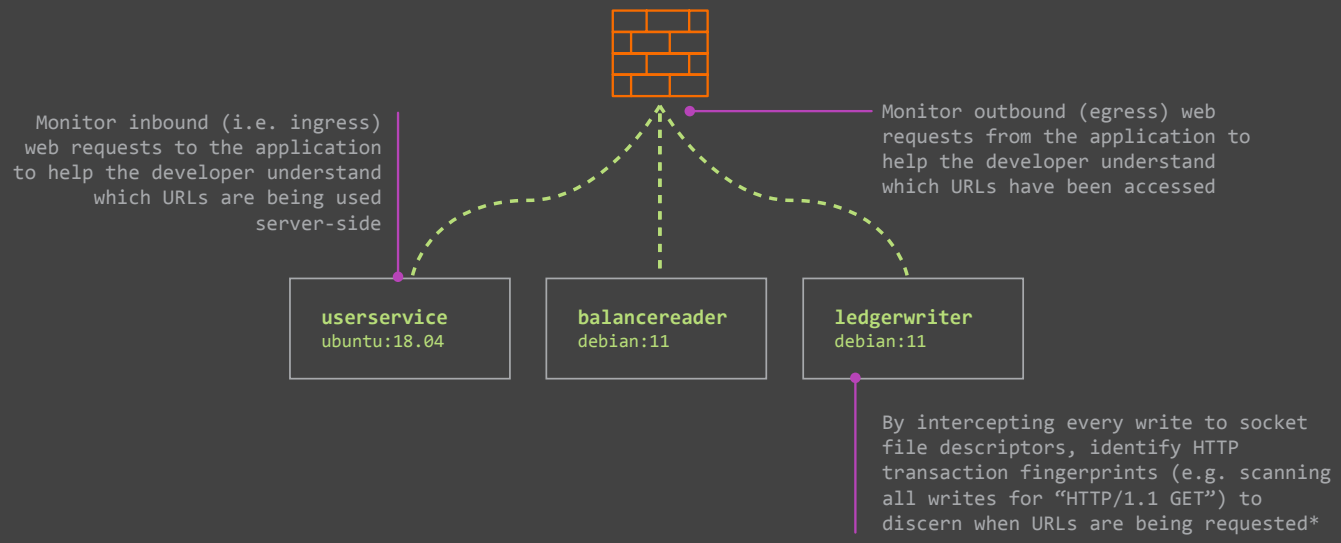
There is a common misconception that having the same library loaded into each process requires more memory than other interception approaches. However, this is generally not true as the operating system will share code and read-only data pages from the LD_PRELOAD library across processes.

Web API Interception

What makes observing web APIs different from what we've described above for other APIs?

When considering how to intercept a Web API (e.g. REST APIs), there are numerous places one can monitor. A key concept in all approaches is first understanding which APIs need to be monitored; these APIs are usually modeled as HTTP/HTTPS GET/POST requests with varying URI patterns.

When monitoring which APIs are called from a client, one might approach the interception task depending on the information required:



// **Figure 4:**
Overview of the various methods for intercepting web APIs

Regardless of which method is chosen, the result is a collection of URIs being requested/served, with the additional capability of calling code inside the API interception tool if the read or write calls are intercepted. Most Web API interception tools will use this information to display which APIs are being requested to the user.

*

When TLS/HTTPS is being used, only import table rewriting and inline patching can be used. In the case of callbacks and eBPF, the data would be monitored after the payload has been encrypted.

// Comparing Interception Techniques

When comparing and evaluating interception techniques, breadth of API coverage is just one of many factors to consider. For example, here are some questions to answer:

- Which technique is easiest to deploy and maintain?

- Which technique has the best performance?

- Which technique is the least likely to affect application behavior/stability?

- Which technique is compatible with the most host systems?

These additional evaluation criteria are especially important when considering the intended use case and target persona. Using API interception to capture application security insights for engineering teams is going to require different characteristics than a tool built to measure application responsiveness. The same could be said of interception tools being purpose-built for engineering teams versus operations—each group wants to intercept different sets of application APIs.

For example:

- Operations teams—who generally want to measure application behavior in live, customer-facing production environments—are going to prefer interception techniques that are stable and performant. However, given the focus on observing application responsiveness on workloads with lengthy uptime, finding a solution with broad API coverage and maintainability can be deprioritized.
- Conversely, during the early stages of the development lifecycle, such as testing in pre-production environments (i.e. staging), extended API coverage and seamless deployment may be worth performance overhead to create a seamless and encompassing experience for developers.

Given the informal analysis of the questions above, and the thorough description of each interception technique in this whitepaper, the following table aims to provide a reasonable estimate of the strengths and weaknesses of each solution:

	Inline Hooking	Import Table Rewriting	Callbacks	Method Rewriting
Performance				
Deployment and Maintenance				
Stability				
Compatibility				
API Breadth				

// Table 3: Comparison of Interception Technologies

When reviewing the above table, it's readily apparent that no single intercept technique can be ranked "Best". This is because the right solution should be found in a combination of the various options. For example, although method rewriting (generally) suffers from sub-par performance, it's the only technology that offers enough code coverage for the creation of a usage-based **Software Bill of Materials (SBOM)**. However, as discussed, using method rewriting exclusively would result in missing APIs of interest. Depending on the use case, this is why method rewriting should be paired with import table rewriting or callbacks for maximum coverage.

// Deepfactor: API Interception and Application Security

Given the increase of cloud native development, accelerated release frequency and application complexity is challenging engineering teams to identify and address security risks without impacting the software development lifecycle. In order to understand and triage cloud native application risks, developers must research data from multiple tools (some static, some interception) to assess risks in custom code, web APIs, open source components, container images, runtime behavior, and sensitive data. The overwhelming number of tools, disruptive instrumentation, and the resulting volume of noisy vulnerability reports delays releases and increases the chances of exposure.

Given these overwhelming challenges, and the industry's drive to adopt DevSecOps, Deepfactor identified an opportunity to develop a next-generation developer security platform. The goal was to intercept and analyze application APIs to provide engineering teams with the information needed to identify, prioritize, and remediate application risks. And with the focus on developers, the experience needed to be seamlessly integrated into the CI/CD pipeline.

In order to accomplish these objectives, Deepfactor employs a number of interception techniques to deliver application-aware security insights with detailed information about application behavior, system calls, and stack traces that help pinpoint vulnerable code. Deepfactor currently uses import table rewriting to intercept APIs in most compiled, dynamically-linked applications. This enables Deepfactor to capture and observe both base and higher level library calls—without significant performance overhead—to create detailed security alerts.

In order to make this easier for developers to implement, Deepfactor instrumentation (i.e. setting the interception library for LD_PRELOAD) can be configured in the following ways:

- **Kubernetes Mutating Webhook** which can automatically load the Deepfactor interception library into Kubernetes Pods upon deployment.
- **"Docker Run"** configuration that automatically launches a containerized application with the Deepfactor interception library preloaded.
- **Standalone CLI tool** wherein a developer can manually run any process with the Deepfactor interception library.

In addition, Deepfactor automatically detects and loads **programming language-specific plugins** for Java, Python, and nodeJS. These language-specific plugins use method rewriting to provide additional contextual information around API usage to the developer. For example, the Java plugin provides Java stack traces for each monitored API, as well as providing a list of all used and unused methods in the application.

Looking to the future, Deepfactor continues to evaluate additional interception techniques to extend platform capabilities, such as improving security insights, identifying risky runtime behaviors, and supporting additional languages and deployment models. If you would like to see the Deepfactor developer security platform in action, you can request a demo here.

{deepfactor}

Deepfactor is a developer security platform that enables engineering teams to quickly discover and resolve security vulnerabilities, supply chain risks, and compliance violations early in development and testing. For more information, follow Deepfactor on **Twitter** or **LinkedIn** or **contact us**.