

XUnprotect: Reverse Engineering macOS XProtect Remediator

Koh M. Nakagawa / FFRI Security, Inc.

2025-10-07

Abstract — The macOS threat landscape has expanded significantly in recent years, with a steady increase in macOS-targeted malware. In response, Apple has extended the capabilities of XProtect by introducing XProtect Remediator (XPR) and XProtect Behavior Service. XPR performs periodic scans to detect and remove malware, but its internal detection and remediation mechanisms have not been well studied.

This paper presents the reverse engineering results of XPR. Because XPR is implemented as stripped Swift binaries, analyzing it is particularly challenging. To address these challenges, we developed custom tools for both static and dynamic analysis. The results show that XPR employs detection logic beyond simple YARA-based scanning, including mechanisms such as the use of Optical Character Recognition (OCR) for detecting Gatekeeper sidestepping techniques. The analysis also revealed insights into Apple-exclusive threat intelligence, including the TriangleDB macOS implant and a publicly unknown Smooth Operator sample. Furthermore, our analysis revealed that XPR's logic is expressed in a custom DSL built with Swift's result builders.

The findings provide technical insights into the internals of XPR. They are relevant for blue teams seeking to understand built-in macOS defenses, for researchers analyzing Apple-exclusive threat intelligence, and for red teams evaluating potential vulnerabilities in XPR.

Chapter 1

Introduction

1.1 Background

With the increasing market share and enterprise adoption of macOS, macOS-targeted malware has grown at an unprecedented rate. This trend is supported by Red Canary, which reports that threat detection for macOS increased by 400% from 2023 to 2024 [1]. In particular, recent years have seen a surge in infostealer-type malware, which has become a major threat on macOS [2], [3]. Furthermore, reported supply-chain attacks by APTs [4], [5] indicate that macOS is no longer a niche target and remains under constant threat.

To counter these threats, Apple has established three layers of malware defense. The Apple Platform Security Guide states the following [6]:

Malware defenses are structured in three layers:

1. Prevent launch or execution of malware: App Store, or Gatekeeper combined with Notarization
2. Block malware from running on customer systems: Gatekeeper, Notarization, and XProtect
3. Remediate malware that has executed: XProtect[Remediator]

XProtect Remediator (XPR), the third layer, detects malware that has already executed on endpoints and remediates infections by removing the malware. XPR comes into play when malware slips past Notarization and Gatekeeper. Notable cases include the 3CX supply-chain attack [4] and infostealers executed by sidestepping Gatekeeper¹ through social engineering [2]. Apple introduced XPR to remove malware that slips past these first two defenses.²

While much research has been conducted on Notarization, Gatekeeper, and traditional XProtect, revealing many aspects of vulnerabilities and internals that enable bypasses [8], [9], [10],

¹This paper differentiates between *bypassing* and *sidestepping* Gatekeeper. Bypassing refers to skipping the Gatekeeper check by exploiting a vulnerability (such as Shlayer [7]), while sidestepping refers to tricking users into ignoring Gatekeeper warnings and running untrusted applications.

²Prior to the introduction of XPR, Malware Removal Tool (MRT) served the same role. As of September 2025, MRT has not been updated since macOS Monterey and has now been replaced by XPR.

research on XPR is limited, and there is little discussion of its detailed internals and vulnerabilities.

1.2 Research Motivation

To address this gap, we reverse engineered XPR to reveal its detailed internals and vulnerabilities. Our motivations fall into the following two categories:

- **Motivation from a defensive perspective:** To clarify the malware families targeted by XPR and how XPR remediates endpoints.
- **Motivation from an offensive perspective:** To identify attacks that exploit its powerful privileges (e.g., entitlements enabling Full Disk Access and execution as root).

Motivation from a Defensive Perspective

Clarifying Malware Families Targeted by XPR

There have been several efforts to identify the malware families targeted by XPR. Alden Schmidt extracted information about file paths and YARA rules included in XPR scanners and identified some of the families associated with each scanner [11]. Building on these results, Phil Stokes identified additional families that Alden had not been able to identify [12] (Figure 1.1).

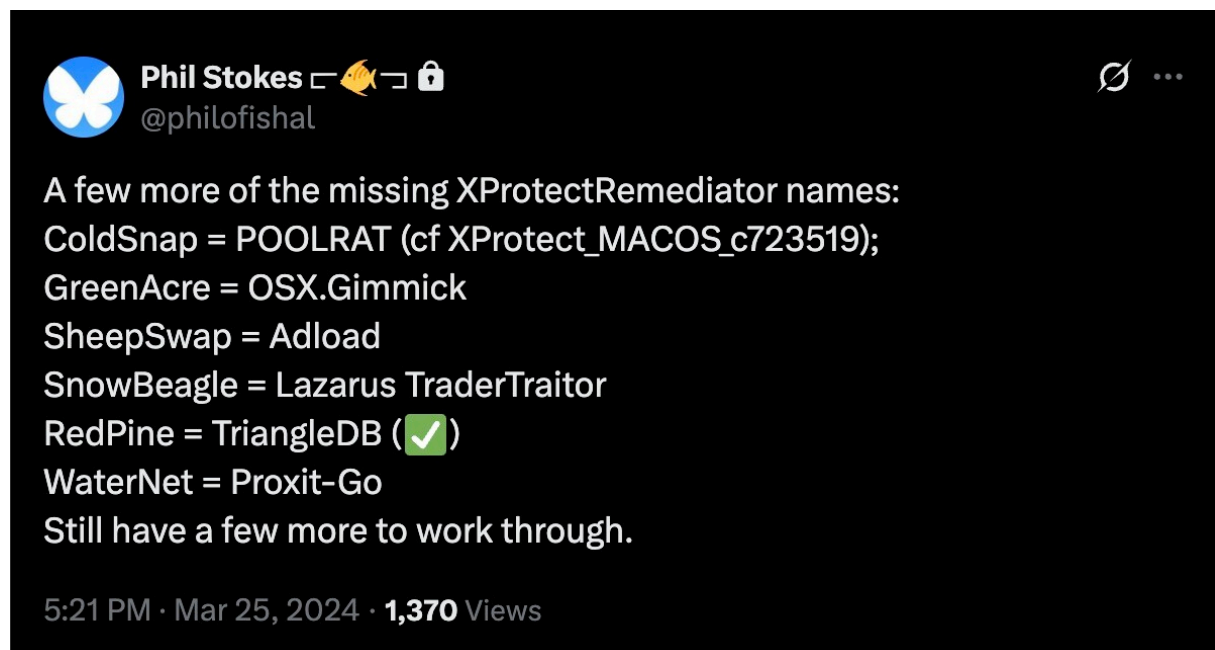


Figure 1.1: This X (formerly Twitter) post by Phil Stokes, published with permission from the author, was posted on March 25, 2024, and discusses malware family names targeted by XPR.

However, not all of XPR’s targets have been revealed. Howard Oakley summarized these research findings in a blog post, noting that several such families remain unknown [13].

Clarifying these families is important for the following reasons:

First, it provides a basis for assessing XPR's defensive capabilities and determining whether additional security measures should be introduced. Knowing which families XPR targets helps identify those that are not covered. This can serve as a basis for decision-making by blue teams considering whether to introduce new security measures.

Second, it can support rapid incident response. Some Antivirus (AV) and Endpoint Detection and Response (EDR) products can collect XPR scanner detection results as telemetry [14]. Clarifying these targets streamlines incident response by allowing responders to map XPR detection results directly to known families and apply established remediation procedures.

Third, it provides an insight into currently active families. XPR represents the final layer of malware defense on macOS. This suggests that XPR may be used as a countermeasure against prevalent malware in the wild. By clarifying XPR's targets, it is possible to gain insight into what families are currently active, helping strengthen countermeasures against those threats.

Clarifying How XPR Remediates Endpoints

The details of how XPR remediates endpoints remain unclear. Each XPR scanner has its own YARA rules [11] and uses them to scan files and remove matching malware [15]. However, simply removing files is unlikely to achieve full remediation, suggesting that other functions are also in place. Clarifying these functions is important, as understanding XPR's remediation capabilities enables responders to distinguish between what is automatically remediated and what requires manual intervention, which is essential for effective and reliable recovery during incident response.

Motivation from an Offensive Perspective

XPR may have vulnerabilities that lead to Transparency, Consent and Control (TCC) bypass. XPR has private entitlements and can access sensitive resources (such as files on the Desktop and Documents folders) even under TCC privacy restrictions. For example, the entitlements for `XProtectRemediatorBlueTop` include two values in version 145: `kTCCServiceSystemPolicyAllFiles` and `kTCCServiceSystemPolicyAppBundles` as keys for `com.apple.private.tcc.allow` (Listing 1.1).

The former is a value that allows Full Disk Access [16], while the latter is a value that allows application modification under TCC App Protection restrictions [17]. Therefore, XPR vulnerabilities could potentially lead to TCC bypass. Indeed, CVE-2024-40842 reported by Gergely Kalman is a vulnerability that enables file content leak with XPR's Full Disk Access privileges [18]. Additionally, XPR runs with both logged-in user and root privileges. Therefore, scanners running with root privileges potentially have privilege escalation vulnerabilities from users to root.

Listing 1.1 This listing shows entitlements of XProtectRemediatorBlueTop.

```
$ codesign -d --entitlements - /Library/Apple/System/Library/CoreServices/XProtect.app/Contents/MacOS/XProtectRemediatorBlueTop
↳ tect.app/Contents/MacOS/XProtectRemediatorBlueTop
Executable=/Library/Apple/System/Library/CoreServices/XProtect.app/Contents/MacOS/XProtectRemediatorBlueTop
↳ cOS/XProtectRemediatorBlueTop
[Dict]
  [Key] com.apple.private.endpoint-security.submit.xp
  [Value]
    [Bool] true
  [Key] com.apple.private.security.signal-exempt
  [Value]
    [Bool] true
  [Key] com.apple.private.security.syspolicy.provenance
  [Value]
    [Bool] true
  [Key] com.apple.private.tcc.allow
  [Value]
    [Array]
      [String] kTCCServiceSystemPolicyAllFiles
      [String] kTCCServiceSystemPolicyAppBundles
  [Key] com.apple.private.xprotect.plugin-service
  [Value]
    [Bool] true
```

1.3 Research Target

We analyzed the following binaries in the application bundle `/Library/Apple/System/Library/CoreServices/XProtect.app`.

- `Contents/MacOS/XProtect`
- `Contents/MacOS/XProtectRemediator<family>` (where `<family>` is the name of the targeted malware family; each of these binaries corresponds to what we refer to as an XPR scanner)
- `Contents/XPCServices/XProtectPluginService.xpc`
- `Contents/Resources/libXProtectPayloads.dylib`

Our primary target is XPR version 145 (update distributed on September 3, 2024). Some analysis includes versions prior to 145, but whenever analysis results from versions other than 145 are presented, this will be explicitly noted.

As of version 145, XPR includes 24 scanners, each implemented as a separate `XProtectRemediator<family>` binary, and each scanner is generally believed to target a specific malware family [19]. For example, `XProtectRemediatorAdload` targets the well-known adware Adload. These family names sometimes align with industry-standard names like Adload, but in many cases they are Apple-specific.

To elucidate both common scanner design and family-specific remediation/detection logic, we analyzed multiple scanners. Findings on common design are presented in Chapter 3, while

findings on family-specific design are detailed in Chapter 4.

The main components related to XPR, including its scanners, are implemented in Swift, as evidenced by section names that include Swift-specific entries such as `__TEXT.__swift5_entry` (Listing 1.2). The symbols in these binaries are stripped.

Listing 1.2 This listing shows Swift-related sections found in XPR binaries.

```
$ otool -l /Library/Apple/System/Library/CoreServices/XProtect.app/Contents/Mach-O/cOS/XProtectRemediatorAdload | grep swift5
↪ sectname __swift5_entry
   sectname __swift5_typerref
   sectname __swift5_capture
   sectname __swift5_reflstr
   ... (snipped for brevity)
```

1.4 Key Findings and Implications

The contribution of this paper lies in elucidating XPR’s internals, including its remediation/detection logic and vulnerabilities, and in presenting implications based on these findings. More detailed findings are explained in subsequent chapters, but the high-level key findings are as follows:

- XPR employs a custom DSL (RemediationBuilder DSL) built with Swift’s result builders to describe remediation/detection logic (Chapter 3).
- XPR leverages the provenance extended attribute to track the origin of files subject to remediation/detection (Chapter 3).
- XPR-targeted malware families not reported in prior work are newly identified (Chapter 4).
- Remediation/detection logic for 15 XPR scanners is unveiled (Chapter 4).
- XPR contains information based on Apple-exclusive threat intelligence (Chapter 4).
- XPR is not necessarily solely for remediation purposes; some scanners perform only detection (no malware removal) and others perform both remediation and detection (Chapter 4).
- XPR contained *elementary* vulnerabilities similar to those seen in EDR/AV products on other platforms (Chapter 5).

These findings provide the following implications:

- **For threat researchers:** Regular analysis of newly added XPR scanners and their differences from existing ones can provide insights into Apple-exclusive threat intelligence. When publishing research, the RemediationBuilder DSL offers a concise way to describe remediation/detection logic (Chapter 4).
- **For blue teams:** An XPR detection result may indicate the presence of a new threat and should prompt deeper investigation. Provenance extended attributes can also serve as valuable forensic artifacts for tracing the origin of threats (Chapter 3 and Chapter 4).

- **For red teams:** New XPR vulnerabilities may be uncovered by examining analogous issues seen in EDR and AV products on other platforms (Chapter 5).

1.5 Organization of This Paper

The remainder of the paper is organized as follows: Chapter 2 describes our methodology for reverse engineering XPR; Chapter 3 presents behaviors common across XPR scanners; Chapter 4 details behaviors specific to each XPR scanner; Chapter 5 reports vulnerabilities we discovered; and Chapter 6 provides a summary of this paper. The Appendix contains supplementary materials, such as Provenance Sandbox internals.

Chapter 2

Methods

This chapter outlines the analysis techniques used in this research. We begin by describing the analysis approach and environment, then discuss the challenges of analyzing Swift binaries and the custom tools we developed to address them. Finally, we cover techniques specific to XPR binaries.

2.1 Analysis Approach & Environment

Our approach combined static analysis with *Binary Ninja* [20] and dynamic analysis with *LLDB* [21]. To support Swift binary analysis, we created custom *Binary Ninja* plugins as well as *LLDB* commands. This chapter introduces these custom tools and a set of commands.

We do not explain language-specific terms related to Swift language features, the compiler, or the standard library. Please refer to the official Swift documentation where needed [22]. Unless otherwise noted, the analysis described here focuses on binaries compiled with Swift 5, where the ABI is stabilized.

The analysis was performed in the following environment. All analysis tools were designed for analyzing x86_64 binaries, and dynamic analysis was carried out on Intel Macs:

- macOS 14.7.3
- Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz

2.2 Challenges in Analyzing Stripped Swift Binaries

As noted in Section 1.3, main components of XPR are implemented in Swift. From the runtime libraries loaded by XPR, we can determine that these binaries are compiled with Swift 5. In addition, Apple ships XPR binaries stripped of symbols, which means the analysis must be performed on stripped Swift binaries.

There are three major challenges when analyzing stripped Swift binaries.

Challenge (1): Lack of Type Information

Unstripped Swift binaries typically retain symbols for type metadata, type metadata accessors, and Protocol Witness Tables (PWTs). These symbols provide valuable clues for understanding which types (such as structs or classes) are instantiated. In the case of PWTs, they also help analyze existential container creation to determine which structs are being assigned to which protocols. Once stripped, however, much of this structured information is lost, making analysis considerably harder.

Challenge (2): Indirect Branch Target Identification

Swift binaries often contain many indirect branches. To perform static analysis, these branch targets usually need to be resolved first through dynamic analysis. When many indirect branches exist, this process quickly becomes cumbersome. The issue is particularly pronounced in code that uses Protocol-Oriented Programming (POP), where program behavior is abstracted through protocols. In such implementations, function calls are frequently dispatched via PWTs, meaning that nearly all of them appear as indirect branches.

Challenge (3): Limitation of *LLDB* Commands

Standard *LLDB* commands (e.g., `expr`) cannot directly dump the elements of an Existential Container and their properties. To examine properties of these elements, we must instead determine the object's memory layout and manually locate the addresses of its properties within that layout.

2.3 Tooling for Analyzing Stripped Swift Binaries

To address the challenges outlined in Section 2.2, we developed two new *Binary Ninja* plugins and a set of custom *LLDB* commands:

- *binja-swift-analyzer* [23] to address Challenge (1)
- *binja-missinglink* [24] and custom *LLDB* commands to address Challenge (2)
- A set of custom *LLDB* commands [25] to address Challenge (3)

binja-swift-analyzer

binja-swift-analyzer is a *Binary Ninja* plugin that analyzes sections containing Swift type information, extracts type-related data (using an implementation based on the `swift-dump` command from the *ipsw* toolkit [26]), and annotates the disassembly listings. Because Swift supports reflection, significant type information is embedded in binaries even when symbols are stripped [27], [28]. For instance, type metadata can be retrieved from nominal type descriptors accessible via `__TEXT.__swift5_types`. The information annotated by this plugin includes:

- type metadata

- type metadata accessors
- PWTs
- class methods

Figure 2.1 shows an example of information annotated by the *binja-swift-analyzer* plugin.

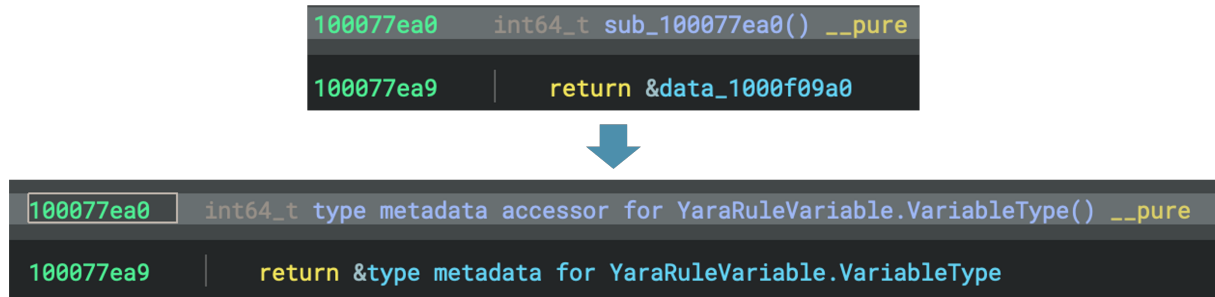


Figure 2.1: This figure shows an example of information annotated by the *binja-swift-analyzer* plugin. A type metadata accessor and type metadata in HLIL output are annotated. HLIL at the top is before running the *binja-swift-analyzer* plugin, and HLIL at the bottom is after running the plugin.

Additionally, the *binja-swift-analyzer* plugin provides the following features to facilitate Swift binary analysis:

- Analyze Swift String objects and annotate referenced strings with comments for immortal large strings¹
- Visualize protocol conformance and class hierarchies

The way Swift String objects are stored in memory depends on their string length. For immortal large strings, the string address is stored in the `_object` property, and the string resides at the address obtained by adding the `nativeBias` value [29], [30]. *binja-swift-analyzer* processes data corresponding to Swift String objects at *Binary Ninja*'s High Level IL (HLIL) level and annotates the corresponding strings when immortal large strings are found (Figure 2.2).

```

100098bef  //
100098bef  // swift_str: /Library/Application Support/ (0x1000d7d60 -
100098bef  // 0x20)
100098bef  sub_10009b4a0(-0x2fffffffffffffe3, -0x7fffffeffff282c0, 9)
100098c02  int64_t rax_6 = sub_10008ac90(sub_10009b430)
100098c14  int64_t rax_7 = sub_10009b3b0(&data_100106988)

```

Figure 2.2: This figure shows HLIL output where addresses of immortal large strings are referenced, with their contents added as comments by the *binja-swift-analyzer* plugin.

The protocol conformance and class hierarchies visualization feature is based on information from the `__TEXT.__swift5_protos` and `__TEXT.__swift5_types` sections. These sections provide details about protocol inheritance, associated type constraints, and class hierarchies. Using this information, the plugin can identify which protocols a struct conforms to, determine class hierarchies, and enumerate structs and classes that conform to specific protocols. Figure 2.3

¹Large strings that do not change their content during runtime are called immortal large strings.

shows an example of protocol conformance and class hierarchies visualized with the *binja-swift-analyzer* plugin.

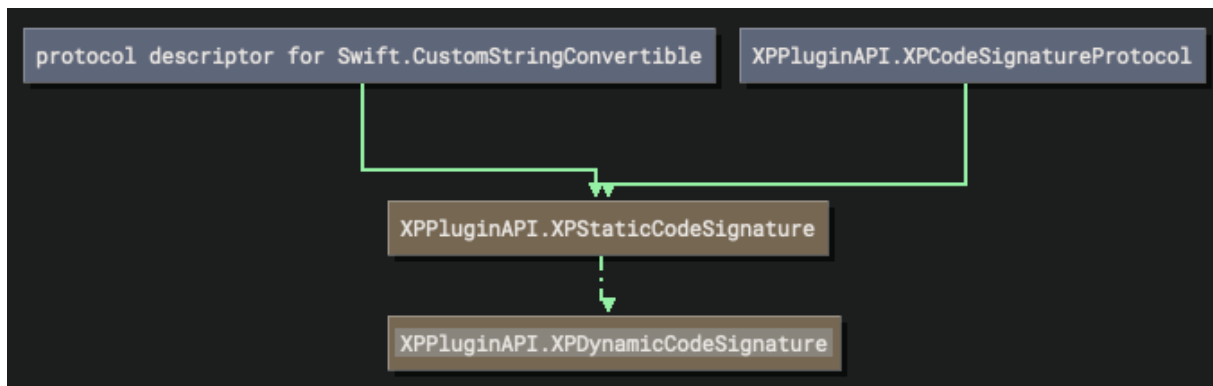


Figure 2.3: This figure shows an example of protocol conformance and class hierarchies visualized through the *binja-swift-analyzer* plugin.

binja-missinglink

binja-missinglink is a *Binary Ninja* plugin that annotates indirect branch targets in disassembly listings using a list of targets collected at runtime. These targets can be collected via two custom *LLDB* commands, `brt_set_bps` and `brt_save` [25].

The `brt_set_bps` command enumerates all indirect branch instructions in the target binary and sets breakpoints at their instruction addresses. Callback functions are set for the breakpoints, which collect information about source and target addresses of branches. This command is implemented using the *LLDB* Scripting Bridge API [31]. After the target program has executed, the `brt_save` command saves the collected branch targets as a JSON file. The *binja-missinglink* plugin then loads this JSON file and annotates the disassembly listings with the resolved branch targets.

In addition, the plugin provides cross-reference functionality, linking indirect branch sources to their targets and vice versa within the disassembly listings. Figure 2.4 shows an example of these cross-reference annotations. Once the JSON file is loaded, indirect branch sources are annotated as `BML_dst: <destination address>` pointing to their targets, and targets are annotated as `BML_src: <source address>` pointing back to their sources.

binja-missinglink not only annotates indirect branch target addresses but also adds related object names for calls through PWTs or vtables. For example, Figure 2.5 shows a function call dispatched via a PWT. In this case, the call goes through the PWT of the `XProtectLaunchdDaemonAgent` class, which conforms to the `XProtectLaunchdDaemonAgentProtocol`.

A Set of Custom *LLDB* Commands for Dumping Existential Containers

An Existential Container is a generic container that can hold any object conforming to a given protocol. Its data layout is fixed and does not vary depending on the protocol. Leveraging

Indirect branch call

10008e3b6				_swift_unknownObjectRetain(r13_9)
10008e3c1				// BML_dst: 0x100067ba0
10008e3c1				int64_t* rax_56 = rbx_14(rax_55, r15_8)
10008e3c9				_swift_unknownObjectRelease(r13_9)

Indirect branch target

100067ba0	int64_t sub_100067ba0(void* arg1 @ r13)
100067ba0	// BML_src: 0x10008e3c1,
100067ba0	// 0x100038ab5(XProtectPluginLaunchdAPI.getLaunchAgents(forUser:)),
100067ba0	// 0x100038ca3
100067ba9	return _swift_retain(*(arg1 + 0x48)) __tailcall

Figure 2.4: This figure shows an example of cross-reference annotation through the *binja-missinglink* plugin.

100099694	void* r15 = *(arg1 + 0x18)
100099698	int64_t r12 = *(arg1 + 0x20)
1000996a2	sub_10009b730(arg1, r15)
1000996b0	// BML_dst: 0x100037e40 (vt:0x1000ef348(pwt of
1000996b0	// XPPPluginAPI.XProtectLaunchdDaemonAgent for
1000996b0	// XPPPluginAPI.XProtectLaunchdDaemonAgentProtocol))
1000996b0	void* rax_5 = (*(r12 + 0x30))(r15, r12)
1000996b5	int64_t i_1 = *(rax_5 + 0x10)

Figure 2.5: This figure shows an example of a function call through PWT annotated by the *binja-missinglink* plugin.

this property, we created custom *LLDB* commands using Swift's `unsafeBitCast` function (see Listing 2.1 for an example custom command). These commands allow the elements of an Existential Container to be directly dumped.

Listing 2.1 This listing shows a custom *LLDB* command for dumping each element of an Existential Container.

```
command regex p_boxed_array 's/(.+)/expr -D 1000 -l Swift -- protocol Empty {};  
↪ unsafeBitCast(%1, to: [any Empty].self)/'
```

This command takes the address of an Existential Container as its argument and applies the `unsafeBitCast` function to cast it into an array of placeholder `Empty` protocol types. After performing this cast and adding type information, Swift runtime can dump each element using its reflection functionality.

Listing 2.2 shows the example output of this command. In this listing, (1) passes the address of the Existential Container as an argument to the `expr` command, while (2) passes the same address to the `p_boxed_array` command. Elements that could not be dumped in (1) are successfully dumped in (2).

2.4 Analysis Techniques Specific to XPR

This section describes analysis techniques specific to XPR binaries.

Symbol Recovery

Although XPR binaries are stripped, some symbols can be recovered by importing those exported by `libXProtectPayloads.dylib`, which is included in `XProtect.app`. This is likely because both `libXProtectPayloads.dylib` and XPR share a common framework in their implementation.

Evidence of this shared framework is demonstrated by comparing the two binaries with *BinDiff* [32]. Figure 2.6 shows the number of functions that are highly similar across the binaries. The comparison revealed over 3,000 functions with a similarity score of 1.0, demonstrating that significant portions of their implementations are identical.

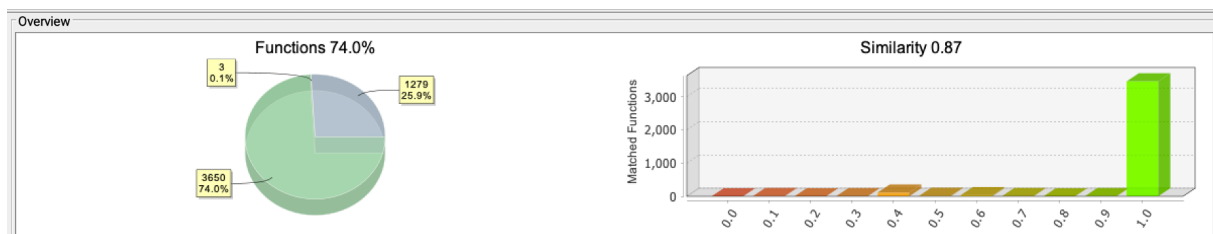


Figure 2.6: This figure shows a *BinDiff* comparison between `libXProtectPayloads.dylib` and XPR.

For this research, we used *Binary Ninja Signature Kit Plugin* [33] to generate a signature library from `libXProtectPayloads.dylib` and then applied it to XPR binaries to support our analysis.

binja-xpr-analyzer

We developed a new *Binary Ninja* plugin specialized for analyzing XPR binaries. For details on its full functionality, see the README in the repository [34]. Here, we highlight one key feature: annotating obfuscated strings in the disassembly listings.

As explained later in Section 3.3.1, file paths and YARA rules used for remediation/detection in XPR binaries are obfuscated and decrypted at runtime. Pointers to these decrypted strings are stored in the `__DATA.__common` section, and during execution, strings are accessed through the pointers stored there.

The *binja-xpr-analyzer* plugin can automatically rename these pointer variables to the corresponding decrypted string names. This significantly improves readability when analyzing code that references decoded strings. Figure 2.7 shows an example of such renamed variables.

1000045ae	int64_t	rule macos_redpine_implant {	strings:	\$classA = "CRConfig"
1000045ae	get_	(
1000045ae)		
1000045bd		return	rule macos_redpine_...ion:	all of them }
1000045be	int64_t	get_	/System/Library/PrivateFrameworks/FMCore.framework()	
1000045cd		return	/System/Library/PrivateFrameworks/FMCore.framework	
1000045ce	int64_t	get_	/System/Library/Frameworks/CoreLocation.framework/CoreLocation()	
1000045dd		return	/System/Library/Frameworks/CoreLocation.framework/CoreLocation	
1000045de	int64_t	get_	/System/Library/Frameworks/AVFoundation.framework/AVFoundation()	
1000045ed		return	/System/Library/Frameworks/AVFoundation.framework/AVFoundation	
1000045ee	int64_t	get_	/usr/lib/libsqlite3.dylib()	
1000045fd		return	/usr/lib/libsqlite3.dylib	

Figure 2.7: This figure shows an example of variables renamed through the *binja-xpr-analyzer* plugin.

Listing 2.2 This listing shows the example output of the `expr` command and the custom `LLDB` command (`p_boxed_array`).

```
(lldb) register read $arg1
rdi = 0x00007ff7bfef190

(lldb) expr -O -l Swift -- 0x00007ff7bfef190 (1)
140702053822864

(lldb) p_boxed_array 0x00007ff7bfef190 (2)
([Empty]) $R0 = 1 value {
  [0] = {
    processConditions = 5 values {
      [0] = {
        constraint = false
        _assess = 0x0000000100084010
        ↪ XProtectRemediatorRedPine`___lldb_unnamed_symbol4201
      }
      [1] = {
        constraint = String (String =
        ↪ "/System/Library/PrivateFrameworks/FMCore.framework")
        _assess = 0x00000001000840c0
        ↪ XProtectRemediatorRedPine`___lldb_unnamed_symbol4208
      }
      [2] = {
        constraint = String (String =
        ↪ "/System/Library/Frameworks/CoreLocation.framework/CoreLocation")
        _assess = 0x00000001000840c0
        ↪ XProtectRemediatorRedPine`___lldb_unnamed_symbol4208
      }
      [3] = {
        constraint = String (String =
        ↪ "/System/Library/Frameworks/AVFoundation.framework/AVFoundation")
        _assess = 0x00000001000840c0
        ↪ XProtectRemediatorRedPine`___lldb_unnamed_symbol4208
      }
      [4] = {
        constraint = String (String = "/usr/lib/libsqlite3.dylib")
        _assess = 0x00000001000840c0
        ↪ XProtectRemediatorRedPine`___lldb_unnamed_symbol4208
      }
    }
    reportOnlyBool = true
    deleteExecutableBool = false
    includePlatformBool = false
  }
}
```


Chapter 3

Common Findings Across Scanners

This chapter describes behaviors observed across multiple XPR scanners. Scanner-specific behaviors are described in the next chapter (Chapter 4).

Before proceeding, we define the terms *remediation* and *detection* used in this paper. *Remediation* refers to identifying malware and removing it from the endpoint. *Detection* refers to identifying malware without removing it. As discussed in the following chapter on scanner-specific results (Chapter 4), some XPR scanners only perform detection, flagging malware without removing it. Others perform remediation alone, and still others perform both detection and remediation. In any case, analytics are transmitted to Apple (if the user opts in).

3.1 Overall Execution Flow

This section outlines the overall execution flow when XPR performs remediation/detection. The flow is also illustrated in Figure 3.1, where each numbered step corresponds to the item listed below. Details of each step are described in subsequent sections.

- (1) `XProtect.app/Contents/MacOS/XProtect` is periodically executed by `launchd` (Section 3.2).
- (2) `XProtect` sends XPC requests to `XProtectPluginService.xpc` (Section 3.2).
- (3) `XProtectPluginService.xpc` invokes each scanner using `NSTask` (Section 3.2).
- (4) Each scanner runs the `mod_init_func0` function, decrypts global strings used for remediation/detection, and performs initialization tasks such as instantiating plugin classes and an `XPAPIHelpers` class (Section 3.3).
- (5) Remediation/detection is performed based on two types of logic (Section 3.4). In some cases, the logic is defined declaratively using the `RemediationBuilder DSL`, while in others it is defined imperatively using `XPAPIHelpers`.
- (6) Before remediating or detecting files in step (5), provenance extended attributes are collected from the target files to identify their originating applications (Section 3.5).

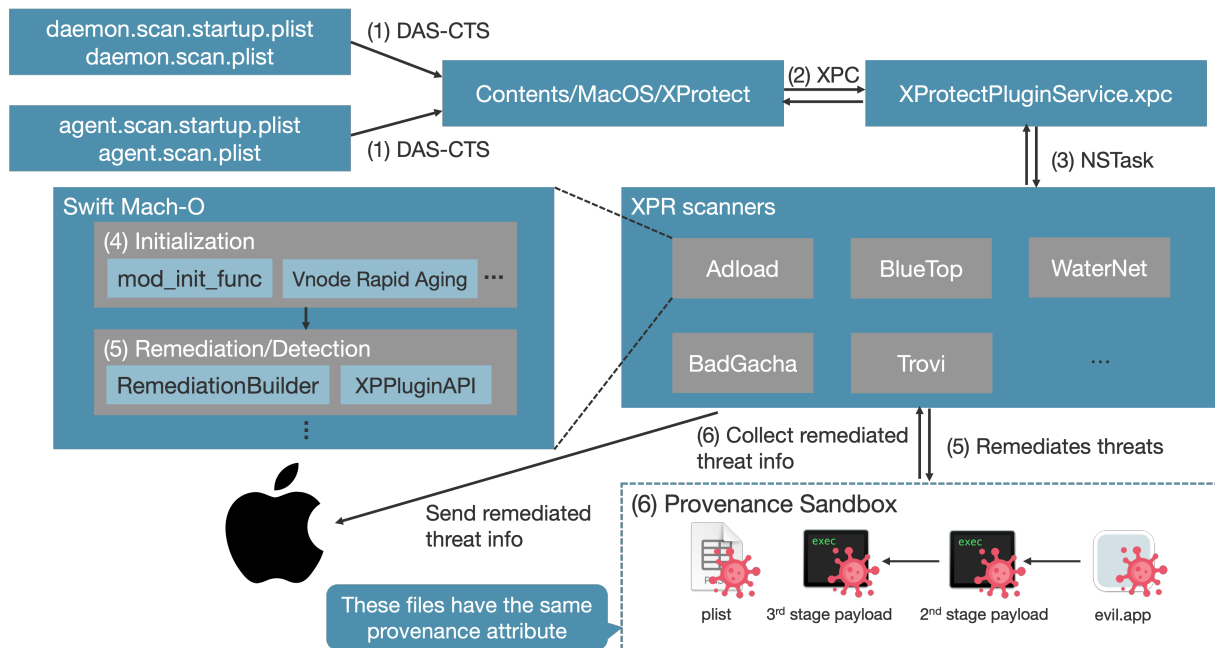


Figure 3.1: This figure shows the overall execution flow of XPR in performing remediation/detection.

3.2 Scheduled Scan

XPR scanners are executed periodically by launchd using two methods:

- **Periodic scan:** Executed as a regularly scheduled task
- **Startup scan:** Executed once when the OS starts up

Both scans are configured through the plist files listed in Listing 3.1. Their presence in both the `LaunchAgents` and `LaunchDaemons` directories indicates that the scans are set up to run under logged-in users as well as root.

Listing 3.1 This listing shows the plist files for configuring XPR periodic and startup scans.

```
/Library/Apple/System/Library/LaunchDaemons/com.apple.XProtect.daemon.scan.startup.plist
↪ rtup.plist
/Library/Apple/System/Library/LaunchDaemons/com.apple.XProtect.daemon.scan.plist
↪ st
/Library/Apple/System/Library/LaunchAgents/com.apple.XProtect.agent.scan.startup.plist
↪ up.plist
/Library/Apple/System/Library/LaunchAgents/com.apple.XProtect.agent.scan.plist
```

Periodic scans are scheduled by DAS-CTS, and their execution timing varies depending on factors such as system resource availability and whether the Mac is connected to external power [35]. Both startup and periodic scans execute the same binary: `XProtect` in `XProtect.app`. As shown in Listing 3.2, the `Program` field specifies `XProtect.app/Contents/MacOS/XProtect`.

When `XProtect` binary runs, it internally calls the `doScan(event:priority:needsCompatInterface:)` function, which sends XPC requests to `XProtectPluginService.xpc`.

Listing 3.2 This listing shows the contents of `com.apple.XProtect.daemon.scan.startup.plist`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  ↪ "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.apple.XProtect.daemon.scan.startup</string>
  <key>Program</key>
  <string>/Library/Apple/System/Library/CoreServices/XProtect.app/Contents/M_
  ↪ acOS/XProtect</string>
  <key>EnvironmentVariables</key>
  <dict>
    <key>START_UP</key>
    <string>1</string>
  </dict>
  <key>EnablePressuredExit</key>
  <true/>
  <key>ProcessType</key>
  <string>Background</string>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

XProtectPluginService.xpc then launches each XPR scanner using `NSTask`. Therefore, all XPR scanners are invoked through XProtectPluginService.xpc.

Although XPR scanners are normally executed through XProtectPluginService.xpc during startup or periodic scans, waiting for these events is unnecessary when performing dynamic analysis. Each scanner can also be run as an independent executable. For example, to reproduce the behavior of the Eicar scanner, the scanner can be executed directly by running `XProtect.app/Contents/MacOS/XProtectRemediatorEicar`. In our analysis, each scanner was executed this way—as a standalone executable.

3.3 Initialization

This section describes initialization procedures common to all XPR scanners.

3.3.1 `mod_init_func0`

Each XPR scanner contains a `mod_init_func0` function¹ (one of the functions in the `_mod_init_func` data section). When executed, this function decrypts global string variables

¹`mod_init_func0` runs before the program entry point and can be registered by applying the `__attribute__((constructor))` attribute to a function.

using a simple XOR cipher. An example of the decryption code in `mod_init_func0` is shown in Listing 3.3.

Listing 3.3 This listing shows a portion of the pseudo-C decompilation of `mod_init_func0` from the RoachFlight scanner.

```
if (data_1000eefb1)
{
    // Address of the string to be decrypted
    int128_t* rax_3 = &data_1000eef88;

    for (int64_t i_1 = 0; i_1 != 0x148; )
    {
        // XOR decryption
        *(uint8_t*)rax_3 ^= (char)(0x303a31323a333000 >> ((uint8_t)i_1 &
            ↪ 0x38));
        i_1 += 8;
        rax_3 += 1;
    }

    data_1000eefb1 = 0;
}
```

Strings decrypted in `mod_init_func0` include the following [36]:

- YARA rules
- File paths
- Regular expressions
- CDHashes
- Process names
- Malware names used internally by Apple (e.g., MACOS.0260dfd)

Pointers to decrypted strings are stored in the `__DATA.__common` section, and scanners access them through these pointers at runtime.

The decrypted strings are used to define the remediation/detection logic of each scanner, and they are critical for identifying the malware families targeted by each scanner. For instance, Alden Schmidt developed a *Binary Ninja* script to decrypt these strings and used the output to identify the targeted malware families [11]. However, Alden's results showed decryption failures for some scanners. To address this, we applied dynamic analysis to successfully decrypt strings across all XPR scanners and published the results, including version-specific differences, in a GitHub repository [37].

3.3.2 Entry Point

At the program's entry point, classes conforming to the `XProtectPluginProtocol` protocol are instantiated. Typically, each scanner instantiates a single plugin class at this stage² (Listing 3.4

²`XProtectRemediatorEicar` and `XProtectRemediatorTrove` are exceptions, each instantiating and executing three plugin classes.

(1)). The arguments passed include the plugin name, a value that appears to represent the plugin class version, and an instance of the `XPPluginStatusCollator` class. The internal details of `XPPluginStatusCollator` were not analyzed in this study, but it is assumed to collect scan results.

Listing 3.4 This listing shows pseudo Swift code for the entry point of the Adload scanner.

```
let adloadPlugin = AdloadPlugin("ADLOAD", 6, XPPluginStatusCollator()) // (1)
adloadPlugin.main(api: XPAPIHelpers.shared) // (2)
```

Next, the `main` method of the plugin class is called (Listing 3.4 (2)), with an instance of the `XPAPIHelpers` class passed as an argument. `XPAPIHelpers` is a helper class that provides access to various types of system information. The functions it offers are described in the next section (Section 3.3.3).³

The primary tasks executed in the `main` method are as follows:

- Instantiate the `XPLoader` class
- Call the `os_signpost` function to measure XPR scan performance
- Unset the `MAGIC` environment variable (a fix for CVE-2024-40842, discovered by Gergely Kalman [18])
- Verify that `XProtectPluginService.xpc` holds the entitlement `com.apple.private.xprotect.trustedplugin-service`
- Enable Vnode Rapid Aging
- Begin remediation/detection

Here, Vnode Rapid Aging disables access-time (`atime`) updates. Enabling this feature is believed to improve scan performance and preserve forensic evidence. Further details on Vnode Rapid Aging are provided in Appendix A.

3.3.3 Details of `XPAPIHelpers` Functionality

This section describes the functionality provided by the `XPAPIHelpers` class. Listing 3.5 shows the properties of the `XPAPIHelpers` class in XPR scanners. These properties were obtained by running the `ipsw swift-dump` command [26].

Table 3.1 lists each property's functionality as shown in their descriptions. These properties fall into two categories: some are protocols, meaning that any class conforming to those protocols can be assigned, while others are concrete classes. Accordingly, our analysis approach differed based on the type of property. For protocol properties, functionality was identified by analyzing the conforming classes and their associated PWT methods. For class properties, functionality was identified by analyzing the methods of the class itself.

³Based on our reverse engineering, the `XPAPIHelpers` class appears to be implemented using the singleton pattern. In the code shown in Listing 3.4, it is therefore accessed as `XPAPIHelpers.shared`. However, because symbols are missing, the exact implementation remains unclear.

Listing 3.5 This listing shows pseudo Swift code for XPAPIHelpers class.

```
class XPAPIHelpers {
    let logger: XLogger
    var pluginService: XProtectPluginDispatchProtocol
    let codeSignature: XProtectPluginCodeSignatureAPIProtocol
    let file: XProtectPluginAPIPath
    var launchd: XProtectPluginLaunchdAPIProtocol
    var launchServices: XPLaunchServicesProtocol
    var yara: XProtectPluginAPIYaraProtocol
    var process: XProtectPluginProcessAPIProtocol
    var event: XProtectPluginAPIEventsProtocol
    let networkSettings: XProtectPluginAPINetworkSettingsProtocol
    var keychain: XProtectPluginKeychainAPIProtocol
    var plugin: XProtectPluginProtocol
    var pipeline: _OBJC_CLASS_$_CPPProfile ?
    var connection: VerifiableXPCCConnectionProtocol
    var configProfiles: XProtectConfigProfilesAPIProtocol
    lazy var alertGUI: XAlertGUIProtocol ?
    var memory: XPPProcessMemoryAPI
    lazy var behavioralEvents: XPEventDatabaseAPIProtocol ??
}
```

Table 3.1: This table shows the properties and functionality provided by the XPAPIHelpers class.

Property Name	Assignable Classes	Key Functionality
logger	XLogger	Logging
pluginService	Unknown	Unknown
codeSignature	XProtectPluginCodeSignatureAPI	Retrieving code signature information and returning as an instance of XPStaticCodeSignature or XPDynamicCodeSignature
file	XProtectPluginAPIPath	File path-related operation functions (e.g., retrieving the HOME directory, creating XPPluginPath instances from file path strings, searching file paths using NSPredicate)
launchd	XProtectPluginLaunchdAPI	Querying launchd services
launchServices	XPLaunchServices	Enumerating registered applications using Launch Services framework
yara	XProtectPluginAPIYara	Creating YaraMatcher instances (YaraMatcher is a class for performing scans using YARA rules)
process	XProtectPluginProcessAPI	Creating an XPPProcess instance from a PID (XPPProcess is a class that provides process-related information)

Property Name	Assignable Classes	Key Functionality
event	XProtectPluginAPIEvents	Unknown (thought to create a BastionProcessEvent instance, but the exact functionality is unknown)
networkSettings	XProtectPluginAPINetworkSettingsAPI ⁴	Retrieving and controlling network settings (e.g., retrieving proxy setting information and changing it)
keychain	XProtectPluginKeychainAPI	Accessing Keychain
plugin	AdloadPlugin, EicarPlugin, etc.	Reference to the plugin class itself
pipeline	Unknown	Unknown
connection	NSXPConnection	Verifying that XProtectPluginService.xpc holds the entitlement com.apple.private.xprotect.trustedpluginservice.
configProfiles	XProtectConfigurationProfiles	Retrieving and modifying configuration profiles (MDM settings, etc.)
alertGUI	XPAlertWindow	Notifying users using NSAlert
memory	XPPProcessMemoryAPI	Accessing process memory
behavioralEvents	XPEventDatabaseAPIs	Accessing XPdb database [39] where XProtect Behavior Service detection results are stored

Here, we highlight two notable properties of XPAPIHelpers and explain them in detail.

configProfiles Property

The configProfiles property accepts classes that conform to the XProtectConfigProfilesAPIProtocol protocol. Currently, only XProtectConfigurationProfiles has been confirmed as a conforming class. Listing 3.6 shows the reconstructed class, obtained from the output of the ipsw swift-dump command.

Listing 3.6 This listing shows pseudo Swift code for XProtectConfigurationProfiles

```
class XProtectConfigurationProfiles: XProtectConfigProfilesAPIProtocol {
    let logger: XPLogger

    func sub_1000195b0 // method (instance)
    func sub_1000199d0 // method (instance)
    func sub_10001a240 // method (instance)
    // <stripped> static func init
}
```

⁴This appears to be a typo by Apple engineers. It should be XProtectPluginAPINetworkSettingsAPI instead of XProtectPluginAPINetworSettingsAPI.

The methods of this class provide functionality for accessing and deleting configuration profiles via the Configuration Profiles framework. For example, they include calls to `_CP_GetUserProfiles` (Figure 3.2). They also invoke functions such as `_CP_RemoveUserProfileWithIdentifier`, which deletes configuration profiles (Figure 3.3).

```
int64_t (* const)() XPPluginAPI.XProtectConfigurationProfiles.sub_10001a240(int64_t arg1 @ r12, void* arg2 @ r13)
10001a24b      void var_f8
10001a24b      void* rsp_1 = &var_f8
10001a252      int64_t var_30 = arg1
10001a25a      id obj_12 = nullptr
10001a26e      id obj = _objc_retainAutoreleasedReturnValue(obj: _CP_GetUserProfiles(&obj_12))
10001a273      id obj_8 = obj_12
10001a27a      int64_t (* const obj_9)()
10001a27a      void* r15
```

Figure 3.2: This figure shows HLIL output that calls `_CP_GetUserProfiles`.

```
int64_t XPPluginAPI.XPConfigurationProfile.sub_100019020(void* arg1 @ r13)
1000193ed      obj_10 = obj_12
1000190ad      else
1000190b3      obj = nullptr
1000190c4      char rax_3 = _CP_RemoveUserProfileWithIdentifier(obj_3, 0, &obj)
1000190d6      obj_10 = _objc_retain(obj)
1000190dc      _objc_release(obj: obj_3)
```

Figure 3.3: This figure shows HLIL output that calls `_CP_RemoveUserProfileWithIdentifier`.

Calling functions from the Configuration Profiles framework requires the `com.apple.private.managedclient.configurationprofiles` entitlement. None of the analyzed scanners were found to hold this entitlement in this study, so the use of this class has not been confirmed. That said, some adware has been reported to install malicious configuration profiles and rewrite DNS server settings [40]. In the future, such malicious profiles could potentially be removed by XPR.

alertGUI Property

The `alertGUI` property accepts classes that conform to the `XPAlertGUIProtocol` protocol. Currently, only `XPAlertWindow` has been confirmed as a conforming class. Listing 3.7 shows the reconstructed class, obtained from the output of the `ipsw swift-dump` command. Because it holds an instance of the `NSAlert` class as a property, it provides functionality for notifying users through dialogs.

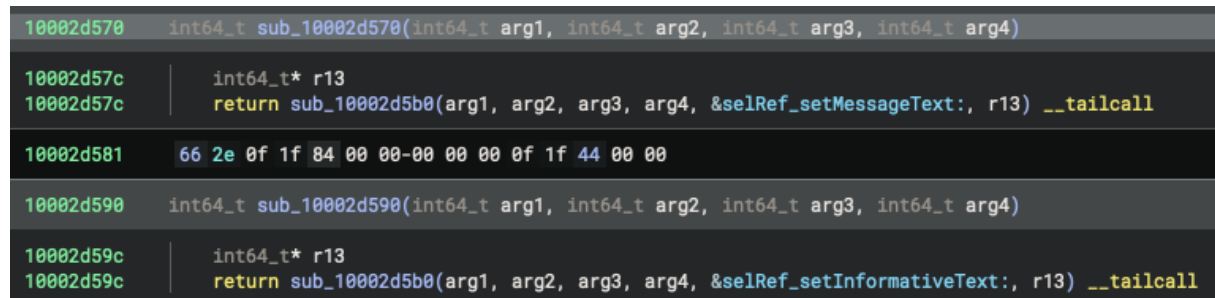
Analyzing the functions registered in the PWT of `XPAlertGUIProtocol` reveals a method that sets the `messageText` and `informativeText` fields of `NSAlert` (Figure 3.4), as well as a method that invokes the `runModal` method. There is also a method that displays an alert window using `NSImageNameCaution` as the icon. Together, these appear to provide functionality for alerting users when XPR remediates/detects threats.

However, this property is not used in the scanners analyzed in this study, and XPR currently does not notify users when threats are remediated or detected.⁵ The presence of the `alertGUI`

⁵To address the fact that XPR does not notify users even when threats are remediated or detected, Howard

Listing 3.7 This listing shows pseudo Swift code of XAlertWindow

```
class XAlertWindow: XAlertGUIProtocol {
    /* fields */
    let alert: NSAlert
    var logger: XLogger
    /* methods */
    // <stripped> func logger.getter
    // <stripped> func logger.setter
    // <stripped> func logger.modify
    // <stripped> static func init
    // <stripped> func method (instance)
    // <stripped> func method (instance)
    // <stripped> func method (instance)
    // <stripped> func method (instance)
}
```



```
10002d570  int64_t sub_10002d570(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4)
10002d57c      int64_t* r13
10002d57c      return sub_10002d5b0(arg1, arg2, arg3, arg4, &selRef_setMessageText:, r13) __tailcall
10002d581      66 2e 0f 1f 84 00 00-00 00 00 0f 1f 44 00 00
10002d590  int64_t sub_10002d590(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4)
10002d59c      int64_t* r13
10002d59c      return sub_10002d5b0(arg1, arg2, arg3, arg4, &selRef_setInformativeText:, r13) __tailcall
```

Figure 3.4: This figure shows HLIL output that sets text in the `messageText` and `informativeText` fields of `NSAlert`.

property suggests that user notification functionality may be added to XPR in the future.

3.4 Remediation/Detection

As explained in Section 3.3.2, XPR performs remediation/detection after enabling Vnode Rapid Aging. To describe the remediation/detection logic, the following steps are required. These steps are implemented primarily through functions provided by various properties of the `XPAPIHelpers` class:

- **Enumeration:** Identifying targets such as files and processes
- **Evaluation:** Assessing whether targets require remediation/detection
- **Execution:** Performing remediation/detection to identified targets

The enumeration and evaluation phases often involve complex conditions. For example, conditions may include file size, whether a Notarization ticket has expired, or the status of a code signature. Although these conditions could be described imperatively in code, doing so tends to reduce maintainability and readability.

Oakley developed XProCheck [41]. This tool allows users to review recent XPR scan results to check for threat detections.

To address this, Apple introduced the RemediationBuilder DSL in XPR. This DSL leverages Swift’s result builders language feature [42], which enables concise, declarative descriptions of complex remediation/detection logic. Result builders are a Swift language feature designed to support the creation of domain-specific languages, best known for powering SwiftUI’s declarative UI syntax. They allow complex logic that collects multiple elements under specific conditions to be expressed concisely. Common use cases include generating HTML and JSON dynamically [43].

It should be noted that not all XPR scanners use the RemediationBuilder DSL. Some scanners define remediation/detection logic imperatively, using the `XPAPIHelpers` class or the lower-level `XPPluginAPI`. Table 3.2 shows which scanners use RemediationBuilder DSL and which do not. This study primarily focused on scanners that utilize the RemediationBuilder DSL.

Table 3.2: This table shows a list of scanners using RemediationBuilder DSL (Yes)/not using (No). Note that the Bundlore scanner was added in XPR version 149, so the result for version 149 is shown.

Scanner Name	RemediationBuilder DSL Usage
XProtectRemediatorAdload	Yes
XProtectRemediatorBadGacha	Yes
XProtectRemediatorBlueTop	No
XProtectRemediatorBundlore	Yes
XProtectRemediatorCardboardCutout	Yes
XProtectRemediatorColdSnap	Yes
XProtectRemediatorCrapyrator	No
XProtectRemediatorDolittle	Yes
XProtectRemediatorDubRobber	No
XProtectRemediatorEicar	Yes
XProtectRemediatorFloppyFlipper	No
XProtectRemediatorGenieo	No
XProtectRemediatorGreenAcre	No
XProtectRemediatorKeySteal	Yes
XProtectRemediatorMRtv3	No
XProtectRemediatorPirrit	Yes
XProtectRemediatorRankStank	Yes
XProtectRemediatorRedPine	Yes
XProtectRemediatorRoachFlight	Yes
XProtectRemediatorSheepSwap	Yes
XProtectRemediatorSnowBeagle	No
XProtectRemediatorSnowDrift	Yes
XProtectRemediatorToyDrop	No
XProtectRemediatorTrove	No
XProtectRemediatorWaterNet	Yes

RemediationBuilder DSL

When creating a DSL with Swift's result builders, it is necessary to define an enum or struct annotated with the `@resultBuilder` attribute. In RemediationBuilder DSL, the enums shown below are defined, and DSLs are created by implementing methods such as `buildBlock` and `buildExpression` within these enums. Separate enums are defined for each remediation/detection target, meaning that RemediationBuilder DSL actually consists of multiple DSLs. Table 3.3 lists the enums defined in the analyzed XPR scanners.⁶

Table 3.3: This table shows the enums defined in XPR scanners.

Enum Name	Description	Collectable Elements
<code>ServiceRemediationBuilder</code>	DSL used to describe launched service remediation/detection conditions	Structs conforming to <code>ServiceCondition</code> protocol
<code>FileRemediationBuilder</code>	DSL used to describe file remediation/detection conditions	Structs conforming to <code>FileCondition</code> protocol
<code>ProcessRemediationBuilder</code>	DSL used to describe process remediation/detection conditions	Structs conforming to <code>ProcessCondition</code> protocol
<code>SafariAppExtensionRemediationBuilder</code>	DSL used to describe Safari App Extension remediation/detection conditions	Structs conforming to <code>SafariAppExtensionCondition</code> protocol
<code>RemediationArrayBuilder</code>	DSL used to combine multiple remediation/detection targets	Structs conforming to <code>RemediationConvertible</code> protocol

We illustrate how the DSL is used with an actual example. Listing 3.8 shows code that defines the remediation logic for the Eicar scanner.

This example uses both `FileRemediationBuilder` and `RemediationArrayBuilder`.

- `FileRemediationBuilder` defines file remediation conditions. It uses structs conforming to the `FileCondition` protocol to specify conditions (block (1) in Listing 3.8). In this case, file at `/tmp/eicar` with a minimum size of 68 bytes that matches the EICAR YARA rule is treated as a remediation target.
- `RemediationArrayBuilder` defines multiple remediation targets. It uses structs conforming to the `RemediationConvertible` protocol to aggregate them (block (2)). In this example only `/tmp/eicar` is defined, but additional remediation targets can be specified in the same block.

The detailed specifications of RemediationBuilder DSL are not covered here. These specifications were determined through reverse engineering and are published in a GitHub repository

⁶Readers may wonder how we determined that the enums in Table 3.3 have the `@resultBuilder` attribute applied, given that the binaries are stripped. Whether XPR truly implements its DSLs using Swift's result builders is not self-evident. This verification is explained in detail in Appendix C.

Listing 3.8 This listing shows pseudo Swift code that describes remediation logic for the Eicar scanner.

```
// This YARA rule is stored in encrypted form and decrypted in the
↳ mod_init_func0 function.
let eicarYara = ""
rule EICAR: Example Test {
    meta:
        name = "EICAR.A"
        version = 1337
        enabled = true
    strings:
        $eicar_substring = "$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!"
    condition:
        $eicar_substring
}
""

EicarRemediator { // (2) Block where the DSL description takes effect via
↳ RemediationArrayBuilder
    File(path: "/tmp/eicar") { // (1) Block where the DSL description takes
↳ effect via FileRemediationBuilder
        // Remediation conditions defined using structs conforming to the
↳ FileCondition protocol
        MinFileSize(68)
        FileYara(YaraMatcher(eicarYara))
    }
    // Additional remediation targets can be defined here if needed
}
```

[\[44\]](#).

3.5 Provenance Extended Attribute

In the remediation/detection phase, XPR also collects provenance extended attributes of target files to identify the applications from which the files originated. Once the originating application is identified, this information is sent as analytics. The provenance extended attribute is known as `com.apple.provenance` and stores an 11-byte integer value. This attribute is automatically attached to files through Provenance Sandbox. In this section, we focus on how XPR uses the provenance extended attribute, without explaining the Provenance Sandbox internals. The mechanism itself is described in [Appendix B](#).

Overview of Provenance Extended Attribute

The following is the minimum background necessary to understand this section:

- (1) `syspolicyd` attaches the provenance extended attribute to an application bundle after Gatekeeper and XProtect verification.

- (2) When the application with a provenance extended attribute is executed, it runs inside the Provenance Sandbox. When that process performs file operations, the same provenance extended attributes from (1) are automatically attached to those files. A list of file operations that result in provenance extended attributes being attached is provided in [Appendix B.2](#).
- (3) The Provenance Sandbox is also inherited by child processes. In other words, when an app running in the Provenance Sandbox spawns a child process, the child process also runs in the same sandbox.
- (4) Application information linked to the value of the provenance extended attribute is recorded in the `provenance_tracking` table of the `/var/db/SystemPolicyConfiguration/ExecPolicy` SQLite3 database. This table includes the application's metadata such as code signature, bundle ID, and team ID.

How XPR Uses Provenance Extended Attributes

XPR retrieves provenance extended attributes from files targeted for remediation/detection. It then sends XPC requests to `syspolicyd` to obtain application information linked to those attribute values. This is handled by the `SPProvenanceTracking` class in the System Policy framework. To send requests to `syspolicyd`, the `com.apple.private.security.syspolicy.provenance` entitlement is required ([Listing 3.9](#)). All XPR scanners have this entitlement.

Listing 3.9 This listing shows entitlements of the BlueTop scanner.

```
$ codesign -d --entitlements - /Library/Apple/System/Library/CoreServices/XProtect.app/Contents/MacOS/XProtectRemediatorBlueTop
↪ tect.app/Contents/MacOS/XProtectRemediatorBlueTop
... (snipped for brevity)
  [Key] com.apple.private.security.syspolicy.provenance
  [Value]
        [Bool] true
... (snipped for brevity)
```

The following Unified log entries are a specific example of XPR using the provenance extended attribute ([Listing 3.10](#)). In this example, the value of the provenance extended attribute associated with *Binary Ninja* is attached to `/private/tmp/eicar`, then the Eicar scanner is executed.

This Unified log entries indicate that application information linked to the value of the provenance extended attribute of `/private/tmp/eicar` is retrieved, including bundle ID and code signature details. The log entries also show that this information is sent as analytics together with details of the deleted `/tmp/eicar` file ([Listing 3.10](#) (1) and (2)).

Why Does XPR Collect Provenance Extended Attributes?

Here we discuss why XPR collects provenance extended attributes and related application information for files targeted for remediation/detection. In the discussion below, *provenance information* refers to both provenance extended attributes and their associated application information.

One possible reason is that provenance information helps identify variants of malware targeted for remediation/detection. We illustrate this point with a specific example. Figure 3.5 shows how Apple might use provenance information under the following assumptions:

- (1) Multiple first-stage malware samples drop the same second-stage payload.
- (2) When XPR scanners are distributed, Apple is unaware of some first-stage malware samples.

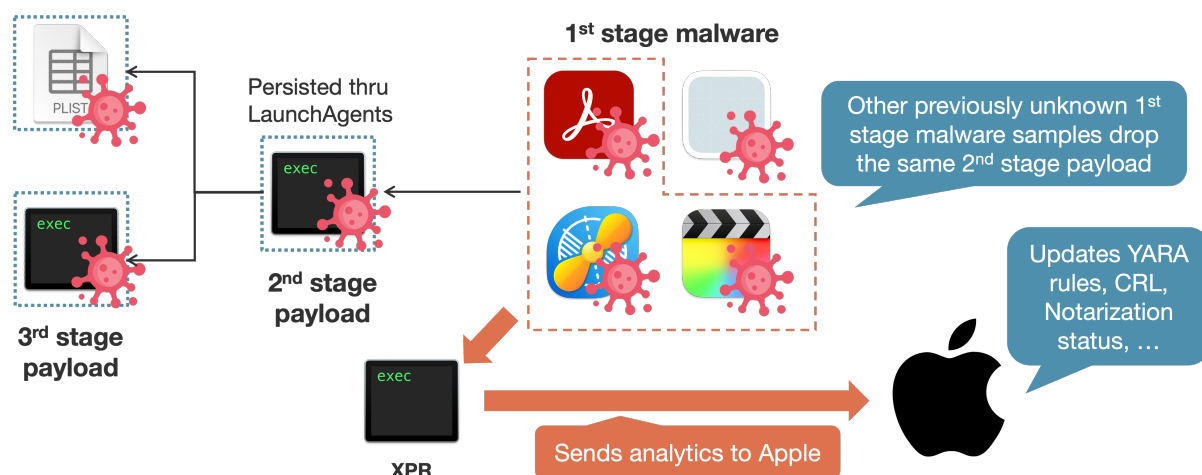


Figure 3.5: This figure shows how Apple might use provenance information.

The first assumption corresponds to cases where same malware is bundled with multiple cracked apps. The second corresponds to newly discovered campaigns where Apple has not yet obtained some first-stage malware samples.

In such cases, by collecting provenance information during second-stage malware remediation and sending it as analytics, Apple can uncover previously unknown first-stage malware. Once identified, Apple can revoke Notarization tickets and update XProtect YARA rules to block execution of the first-stage malware and limit further spread.

Provenance information is also valuable to third-party security vendors and blue teams. For example, it can help identify applications that achieve persistence or those that use reflective loaders via `NSLinkModule` and `NSCreateObjectFileImageFromMemory`. Specific use cases are described in Appendix B.

Listing 3.10 This listing shows Unified log entries when the Eicar scanner uses the provenance extended attribute of a remediation target.

```

Info          0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:VerifiableXPCCConnectionProtocol]
↳ Connecting to service PID 606 (entitled: true, Apple Signed: true
Default      0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:YaraDispatch] Initialized libYARA
↳ version 3.11
Info          0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:YaraRule] Adding
↳ YaraRuleVariable[path] to compiler
Default      0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:YARARuleMatchV4] Rule
↳ YaraRule[EICAR]: Hit
Debug        0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:PathAPI] Deleted Path[/tmp/eicar]
Info          0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:XPEvent] Sending analytics -
↳ com.apple.bastion.file-provenance-info: ["provenance_cdhsh":
↳ 572d5afff571131d3e00ec5b349d081057bc24f7, "provenance_signingid":
↳ com.vector35.binaryninja, "plugin_identifiier": Eicar.User, "file_sha256":
↳ 131f95c51cc819465fa1797f6ccacf9d494aaaff46fa3eac73ae63ffbd8267,
↳ "provenance_teamid": 6ZMTJ2Y7Q2, "provenance_bundleid":
↳ com.vector35.binaryninja, "bastion_version": 145, "plugin_version": 2,
↳ "event_timestamp": 1758452604.232873, "bastion_pipeline_id":
↳ F2D715AC-7AFF-4C84-B562-2B0DBC76CE5, "provenance_filename":
↳ /Applications/Binary Ninja.app] (1)
Info          0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:XPEvent] Sending analytics -
↳ com.apple.bastion.path: ["file_sha256":
↳ 131f95c51cc819465fa1797f6ccacf9d494aaaff46fa3eac73ae63ffbd8267,
↳ "date_created": 1758452501.589466, "event_type": path_delete,
↳ "filesystem_path": /tmp/eicar, "bastion_version": 145,
↳ "signature_is_valid": 0, "status_code": Success, "mime_type": text/plain;
↳ charset=us-ascii, "plugin_identifiier": Eicar.User, "bastion_pipeline_id":
↳ F2D715AC-7AFF-4C84-B562-2B0DBC76CE5, "plugin_version": 2,
↳ "notarization_status": 0, "quarantined": 0, "file_size": 69,
↳ "status_message": Success, "event_timestamp": 1758452604.234472,
↳ "date_modified": 1758452501.589843] (2)
Info          0x0          3507    0    XProtectRemediatorEicar:
↳ [com.apple.XProtectFramework.PluginAPI:PluginStatusCollator] Collated
↳ message: Optional(["Success - File: Path[\\tmp\\eicar\\"]")

```

Chapter 4

Scanner-specific Findings

This chapter describes behaviors specific to each XPR scanner. We begin with Table 4.1, which lists the scanners analyzed in this study and provides links to the sections where each is described in detail. Our analysis primarily focused on scanners that employ the Remediation-Builder DSL.

Not all scanners are discussed in this chapter. Instead, we highlight those that exhibit particularly interesting behaviors or target malware families that have received limited coverage in public reports. This chapter presents the analysis results for individual scanners and, finally, concludes with key findings derived from analyzing multiple scanners and their implications. The full set of analysis results is available in our GitHub repository [45].

Table 4.1: This table summarizes the analysis results for each scanner. For malware families like Adload, Bundlore, and Pirrit, many public reports are available, so only representative ones are included here. Entries marked “Generic” indicate scanners that, according to our analysis, use generic detection logic rather than targeting a specific malware family. The Bundlore scanner was added in XPR version 149 and analyzed in that version.

Scanner Name	Targeted Malware Family	Related Reports	Corresponding Section
Adload ¹	Adload	[46], [47]	Section 4.1
BadGacha ²	Generic	N/A	Section 4.2
Bundlore ³	Bundlore	[48]	N/A
CardboardCutout ⁴	Generic	N/A	Section 4.6
ColdSnap ⁵	POOLRAT	[4]	N/A
Dolittle ⁶	Genieo variant (MaxOfferDeal)	N/A	N/A
Eicar ⁷	EICAR	N/A	N/A
KeySteal ⁸	KeySteal	[49]	N/A
Pirrit ⁹	Pirrit	[50], [51]	N/A
RankStank ¹⁰	Smooth Operator	[4]	Section 4.3
RedPine ¹¹	TriangleDB	[52]	Section 4.5
RoachFlight ¹²	Smooth Operator	[4]	Section 4.4

Scanner Name	Targeted Malware Family	Related Reports	Corresponding Section
SheepSwap ¹³	Adload variant	[53]	Section 4.7
SnowDrift ¹⁴	CloudMensis	[54]	N/A
WaterNet ¹⁵	Adload variant	[55]	N/A

4.1 XProtectRemediatorAdload

XProtectRemediatorAdload has been included since the initial release of XPR and is responsible for remediating the well-known Adload adware. The scanner underwent significant changes in XPR version 131 (April 2024) [56], when its YARA rules were completely rewritten. As of version 145, it contains 86 YARA rules totaling more than 1,400 lines [37].

Because Adload has already been extensively analyzed by various researchers [46], [47], our discussion of its remediation/detection logic focuses on aspects that have received little public attention. In particular, we highlight the distinctive Proxy Remediation feature unique to this scanner, as well as its inclusion of `~/ .crontab` as a scanning target—an element that has not been mentioned in public write-ups on Adload.

4.1.1 Proxy Remediation

Listing 4.1 shows the remediation logic for XProtectRemediatorAdload expressed in RemediationBuilder DSL.

In (1) of Listing 4.1, `followUpRemediation` is defined, and `ProxyRemediation` is added within it. This `followUpRemediation` specifies additional remediation steps to be executed when the Service conditions shown in (2) are met. In this case, it removes local proxy settings that match certain hosts and ports.

This proxy remediation is believed to have been added to remove local proxies configured by Adload. According to Phil Stokes, Adload has been observed setting a SOCKS proxy at

¹<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/Adload>

²<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/BadGacha>

³<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/Bundlore>

⁴<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/CardboardCutout>

⁵<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/ColdSnap>

⁶<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/Dolittle>

⁷<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/Eicar>

⁸<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/KeySteal>

⁹<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/Pirrit>

¹⁰<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/RankStank>

¹¹<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/RedPine>

¹²<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/RoachFlight>

¹³<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/SheepSwap>

¹⁴<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/SnowDrift>

¹⁵<https://github.com/FFRI/XPRTTestSuite/tree/main/TestSuite/WaterNet>

Listing 4.1 This listing shows pseudo Swift code representing the remediation logic of the Adload scanner.

```
let adloadYara = """
... // snipped for brevity
"""

... // snipped for brevity

AdloadRemediator {
    Service(tag: nil) { // (2)
        ServiceExecutable {
            FileMacho(true)
            FileNotarised(false)
            FileYara(YaraMatcher(adloadYara))
        }
    }
    .followUpRemediation(ProxyRemediation(reportOnly: false, hosts:
        ↪ ["localhost", "0.0.0.0", "::1", "127.0.0.1"], ports: [8080])) // (1)
    .onMatchCallbacks([func_0x000000001000901e0, func_0x000000001000901e0])
    ... // snipped for brevity
}
```

localhost:8080 [57]. By executing the proxy remediation defined in (1), local proxies installed by Adload can be removed.

The proxy remediation feature is implemented through the `XProtectPluginAPINetworkSettingsAPI` class. This class leverages the System Configuration framework to retrieve and modify network settings on the endpoint.

4.1.2 Remediation of Executables Listed in `~/ .crontab`

The remediation/detection logic of `XProtectRemediatorAdload` includes analyzing the crontab file at `~/ .crontab` and removing executables registered there if they meet the conditions shown in (1) of Listing 4.2.

To our knowledge, the creation of `.crontab` files in the home directory by Adload has not been documented in public write-ups. Moreover, for persistence using crontab, the files must be stored in `/private/var/at/tabs/` [58]. At present, detailed information about this behavior is lacking, and the reason why `XProtectRemediatorAdload` includes such remediation logic remains unclear.

4.2 XProtectRemediatorBadGacha

`XProtectRemediatorBadGacha` was added in XPR version 91 (March 2023). It is notable for containing decrypted strings, such as `.background`, `right-click`, and `click open`, which appear unrelated to detection (Listing 4.3).

Listing 4.2 This listing shows pseudo Swift code representing the remediation logic of the Adload scanner.

```
let adloadYara = """
... // snipped for brevity
"""

... // snipped for brevity

AdloadRemediator {
    ... // snipped for brevity

    // enumCrontabExecutables is a function that parses the crontab file and
    ↪ returns a list of registered executables
    // Note: function symbols of XProtectRemediatorAdload are stripped, so the
    ↪ function name does not express the actual function name
    for file in enumCrontabExecutables("~/crontab") {
        File(path: file) { // (1)
            FileMacho(true)
            FileNotarised(false)
            FileYara(YaraMatcher(adloadYara))
        }
    }
}
```

The BadGacha scanner performs two independent scans. Both scans are detection-only, meaning they do not remove threats even when threats are found.

- Detection of Gatekeeper sidestepping using Optical Character Recognition (OCR)
- Detection of processes without backing files

Notably, detection of processes without backing files was removed in XPR version 135. The following sections describe the behavior of these two scans in detail.

4.2.1 Detection of Gatekeeper Sidestepping Using OCR

Before detailing the scan logic, we briefly review a technique employed by some malware to sidestep Gatekeeper. Specifically, this technique commonly leverages a DMG file. A DMG file intended for application installation often includes a background image containing installation instructions; attackers can embed directions in this image that tell users to ignore Gatekeeper warnings and execute the application, thereby abusing the DMG background to sidestep Gatekeeper.

Figure 4.1 shows one such example from AMOS [59]. The background image instructs users to “right-click” the bundled application in the DMG and select “Open,” thereby launching it while ignoring Gatekeeper warnings.

A background image used for installation guidance is embedded in a DMG file, with its configuration stored in the `.DS_Store` file [60]. Typically, this image is saved as a hidden file or in a

Listing 4.3 This listing shows decrypted strings in the BadGacha scanner.

```
.background
.background.
right-click
right click
option click
choose open
click open
press open
unidentified developer
are you sure you want
will always allow it
run on this mac
```

dedicated `.background` folder.

The BadGacha scanner targets background images that contain strings indicative of Gatekeeper sidestepping. It performs detection using the following steps:

- Enumerate currently mounted DMG files using `FileManager.mountedVolumeURLs`
- Extract background images from these DMGs, targeting the following locations:
 - Images in the `.background` directory at the DMG root
 - Images with the `.background.` prefix at the DMG root
- Apply OCR to the extracted images and check for Gatekeeper sidestepping-related strings such as “right click” and “click open”
 - String detection via handlers passed to `VNRecognizeTextRequest`’s `initWithCompletionHandler`

As noted earlier, this scan performs detection only. No remediation is attempted—for example, DMG files are neither unmounted nor deleted.

4.2.2 Detection of Processes Without Backing Files

This detection logic is expressed in RemediationBuilder DSL, as shown in Listing 4.4. Since `reportOnly()` is specified in (1), even when processes are detected, they are not terminated but merely reported.

Listing 4.4 This listing shows pseudo Swift code that defines the detection logic of the BadGacha scanner for processes without backing files.

```
BadGachaRemediator {
    Process {
        ProcessHasBackingFile(false)
    }.reportOnly() // (1)
}
```

Here, `ProcessHasBackingFile` is set to `false`, meaning that processes without backing files are flagged as detection targets. This logic is believed to address macOS malware that per-

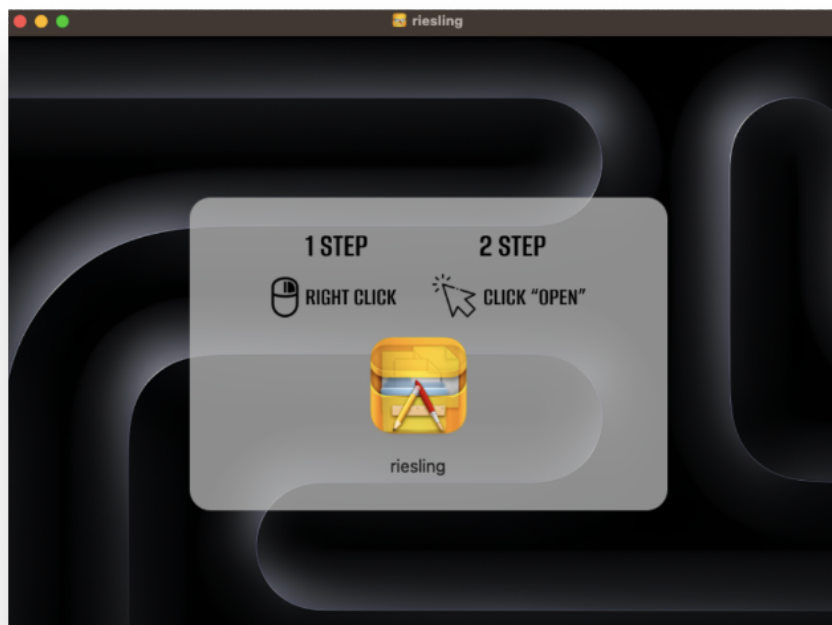


Figure 4.1: This figure shows a background image of a DMG file used by AMOS [59].

forms self-deletion [4]. However, it is also known to generate many false positives. Processes without backing files often occur temporarily during software updates, making them likely to be misclassified. Probably in response to numerous false-positive reports (such as a report on Apple’s forums [61]), this logic was removed from the BadGacha scanner in version 135.

4.2.3 Considerations on Malware Targeted by the BadGacha Scanner

The two scans performed by the BadGacha scanner are generic and unlikely to target specific malware families. In particular, the OCR-based scan is known to flag DMG files associated with multiple malware families, including:

- ChromeLoader [62]
- Empire Transfer [63]
- AMOS [59]
- Cuckoo [64]

In addition, both scans are detection-only and do not perform any remediation; they transmit the detection results to Apple as analytics. Given these factors, the BadGacha scanner appears to function primarily as a threat-hunting scanner, designed to detect DMG files leveraged by currently prevalent malware.

4.3 XProtectRemediatorRankStank

XProtectRemediatorRankStank was added in XPR version 96 (April 2023). XProtectRemediatorRoachFlight was introduced at the same time and is discussed in the next section (Section 4.4). The RankStank scanner is known to remove the Smooth Operator malware [11].

This scanner targets both the first-stage and second-stage payloads of Smooth Operator. The remediation logic expressed in RemediationBuilder DSL is shown in Listing 4.5.

Listing 4.5 This listing shows pseudo Swift code that defines the remediation logic for the RankStank scanner.

```
let rankStankYara = ""
... // snipped for brevity
""

RankStankRemediator {
    File(path: "/Applications/3CX Desktop App.app/Contents/Frameworks/Electron
↪ Framework.framework/Versions/A/Libraries/libffmpeg.dylib") { // (1)
        FileYara(YaraMatcher(rankStankYara))
    }
    File(predicate: NSPredicate(format: "kMDItemDisplayName ==
↪ 'libffmpeg.dylib'")) { // (2)
        FileYara(YaraMatcher(rankStankYara))
    }
    File(path: "~/Library/Application Support/3CX Desktop App/.main_storage")
↪ {}
    File(path: "~/Library/Application Support/3CX Desktop App/UpdateAgent") {}
↪ // (3)
}
```

In (1) of Listing 4.5, backdoored libffmpeg.dylib is specified for deletion. (2) defines logic intended to delete the libffmpeg.dylib file even when the 3CX Desktop App is not installed under /Applications, using NSPredicate.

However, the current DSL does not support file deletion via NSPredicate, so (2) is non-functional.¹⁶ As a result, the RankStank scanner cannot delete libffmpeg.dylib if 3CX Desktop App is installed outside /Applications.

In (3), the second-stage payload UpdateAgent is specified for deletion. As explained in the next section, the RoachFlight scanner also targets second-stage payloads but terminates their running processes. According to Patrick Wardle's report, UpdateAgent implements self-deletion, leaving no files on disk if execution succeeds [4]. Why explicit file deletion is included remains unclear, though possible explanations are that self-deletion may be interrupted by EDR products or that variants exist without this functionality.

¹⁶Although a NSPredicate instance is created and passed as an argument, it will be ignored in the File's initializer.

4.4 XProtectRemediatorRoachFlight

This scanner was introduced in the same update as the RankStank scanner. It removes Smooth Operator's second-stage payloads. The remediation logic expressed in RemediationBuilder DSL is shown in Listing 4.6.

Listing 4.6 This listing shows pseudo Swift code that defines the remediation logic for the RoachFlight scanner.

```
let targetCDHashes = ["04e23817983f1c0e9290ce7f90e6c9e75bf45190",
↳ "99c31f166d1f1654a1b7dd1a6bec3b935022a020"]

RoachFlightRemediator {
    for cdHash in targetCDHashes {
        Process {
            ProcessCDHash(cdHash)
        }.deleteExecutable()
    }
}
```

From Listing 4.6, processes with two CDHashes are specified for deletion. Among them, 04e23817983f1c0e9290ce7f90e6c9e75bf45190 corresponds to the CDHash for the known second-stage payload (Listing 4.7), with detailed analysis provided by Patrick Wardle [4].

Listing 4.7 This listing shows the CDHash for the known second-stage payload of Smooth Operator.

```
$ shasum -a 256 /tmp/UpdateAgent
6c121f2b2efa6592c2c22b29218157ec9e63f385e7a1d7425857d603ddef8c59
↳ /tmp/UpdateAgent
$ codesign -dvvvv /tmp/UpdateAgent 2>&1 | grep CDHash=
CDHash=04e23817983f1c0e9290ce7f90e6c9e75bf45190
```

When investigating Smooth Operator–related CDHashes based on public information, no sample corresponding to 99c31f166d1f1654a1b7dd1a6bec3b935022a020 was found. This CDHash is therefore considered to represent an unknown UpdateAgent variant. The rationale is that the known UpdateAgent sample (04e23817983f1c0e9290ce7f90e6c9e75bf45190) has limited functionality: it only sends device information to C2 servers then exits. It is reasonable to assume that attackers also prepared and distributed variants with more complex functionality for truly targeted systems. Patrick Wardle likewise noted the possibility of other second-stage payload variants [4].

Another notable aspect of this Smooth Operator remediation is the division of functionality between the RoachFlight scanner and the RankStank scanner. Why was the process-deletion logic of the RoachFlight scanner not implemented in the RankStank scanner, but instead separated into a dedicated scanner handling only UpdateAgent process deletion?

One hypothesis is that Apple intended to facilitate triage of systems that were true targets of the attackers. While exact figures are unclear, over 600,000 companies are reported to use

the 3CX Desktop App [65]. In supply chain attacks, only a small fraction of these systems are believed to be true targets. By dividing functionality between the RankStank scanner and the RoachFlight scanner, Apple may have sought to better distinguish truly targeted systems.

4.5 XProtectRemediatorRedPine

XProtectRemediatorRedPine was added in XPR version 114 (October 2023) and removed in version 142 (August 2024). It is the only scanner that has been removed from XPR. Our analysis is based on version 141.

Decrypted strings include the YARA rule shown in Listing 4.8. This rule matches samples of the TriangleDB iOS implant reported by Kaspersky researchers [52].

Listing 4.8 This listing shows decrypted strings in the RedPine scanner.

```
rule macos_redpine_implant {
  strings:
    $classA = "CRConfig"
    $classD = "CRPwrInfo"
    $classE = "CRGetFile"
    $classF = "CRXDump"
  condition:
    all of them
}

/System/Library/PrivateFrameworks/FMCore.framework
/System/Library/Frameworks/CoreLocation.framework/CoreLocation
/System/Library/Frameworks/AVFoundation.framework/AVFoundation
/usr/lib/libsqlite3.dylib
```

Kaspersky researchers suggested that macOS may also have been targeted by Operation Triangulation [52]. Accordingly, the RedPine scanner is considered a scanner for detecting TriangleDB macOS implants. Although the macOS implant samples are not publicly available at this time, analyzing this scanner provides insights into them.

When run as root, the RedPine scanner performs two separate scans:

- Scan the memory of running processes (excluding platform processes) and check whether the main executables match the YARA rule shown in Listing 4.8 (referred to as the *In-memory Scanner* below)
- Examine loaded libraries and check whether any of their file paths exactly match those listed in Listing 4.8 (referred to as the LoadedLibrary Scanner by Apple)

Note that the RedPine scanner only performs detection through these two scans and does not terminate the implant processes even if they are detected. Why did Apple choose not to remediate the implant processes? One possibility is that Apple deliberately deployed the RedPine scanner to identify devices targeted by Operation Triangulation and to facilitate investigation of macOS implants on infected systems. As explained in the next section (Section 4.5.1), these implants were likely deployed in memory, and terminating their processes would result in loss

of forensic evidence, making analysis more difficult. To preserve such evidence, Apple likely restricted the scanner to detection only, and when detections occurred, handled cases individually by notifying affected users through procedures for advanced spyware victimization [66].

4.5.1 In-memory Scanner

The in-memory scanner examines the `__TEXT` segment of the main executables of running processes and checks whether they match the YARA rule shown in Listing 4.8. Since the RedPine scanner has the `com.apple.system-task-ports.read` entitlement, it can call the `task_read_for_pid` function and obtain task read ports for all processes, even in a System Integrity Protection (SIP)-enabled environment.

An important point is that other scanners do not have the `com.apple.system-task-ports.read` entitlement and therefore cannot read the memory of running processes. As a result, they typically check whether YARA rules match against backing files. The RedPine scanner differs in that it checks memory contents directly.

One explanation for this design is that macOS implants may have been deployed in memory, similar to the iOS implants [52] (see quote below). In such cases, scanning backing files would not have been possible, making direct memory scanning necessary.

The implant, which we dubbed TriangleDB, is deployed after the attackers obtain root privileges on the target iOS device by exploiting a kernel vulnerability. It is deployed in memory, meaning that all traces of the implant are lost when the device gets rebooted.

4.5.2 LoadedLibrary Scanner

The LoadedLibrary Scanner is expressed in RemediationBuilder DSL, as shown in Listing 4.9:

Listing 4.9 This listing shows pseudo Swift code that defines the remediation logic for the LoadedLibrary Scanner in the RedPine scanner.

```
RedPineScanner {
  Process {
    ProcessIsAppleSigned(false)
    HasLoadedLibrary("/System/Library/PrivateFrameworks/FMCore.framework")
    HasLoadedLibrary("/System/Library/Frameworks/CoreLocation.framework/Co
↵ reLocation")
    HasLoadedLibrary("/System/Library/Frameworks/AVFoundation.framework/AV
↵ Foundation")
    HasLoadedLibrary("/usr/lib/libsqlite3.dylib")
  }.reportOnly()
}
```

Since `ProcessIsAppleSigned` is set to `false`, the scanner targets processes not signed by Apple. It also checks whether the specified libraries are loaded, using the file paths listed in Listing 4.8 with `HasLoadedLibrary`.

A notable aspect of Listing 4.9 is that the dylibs are specified using their symbolic link paths rather than the actual runtime-resolved paths. On macOS, the CoreLocation and AVFoundation file paths are symbolic links that resolve at runtime to the dylibs in the Versions/Current directory of each framework. Consequently, the correct parameters for HasLoadedLibrary should be the resolved paths in Versions/Current. For example, for CoreLocation, the appropriate path is shown in Listing 4.10:

Listing 4.10 This listing shows the resolved file path of the CoreLocation symbolic link.

```
/System/Library/Frameworks/CoreLocation.framework/Versions/Current/CoreLocation
```

Another unusual point is that the FMCore.framework file path in Listing 4.9 refers to a directory. Obviously, directories cannot be loaded as dylibs. Why, then, are such non-dylib paths specified? Several hypotheses may explain this:

- (1) Incorrect file paths are specified by Apple.
- (2) macOS implants bypassed Signed System Volume (SSV) and SIP filesystem protection and directly overwrote two system symbolic links or a directory and the FMCore.framework directory.
- (3) macOS implants implemented custom reflective loaders.

Hypothesis (1) cannot be ruled out but seems relatively unlikely. On iOS, the CoreLocation and AVFoundation file paths point directly to dylib entities rather than symbolic links, so mis-specification is plausible. However, FMCore.framework is also a directory on iOS, making such a basic mistake improbable.

Hypothesis (2) is also unlikely. Directly overwriting two symbolic links or a directory would require bypassing both SIP filesystem protection and SSV integrity verification performed during OS boot. Moreover, platform binaries depending on FMCore.framework would fail to execute normally, destabilizing the system, which makes this scenario improbable.

Here, we discuss details of Hypothesis (3) in the next section.

4.5.3 Custom Reflective Loader

TriangleDB iOS implants are reported to have implemented reflective loaders that load modules directly into memory [52]. macOS implants likely employed similar functionality.

As Patrick Wardle has pointed out, when dylibs are loaded through reflective loaders, their backing files often appear empty—an indicator that can be used to detect such loaders [67]. But what if arbitrary file paths could be assigned as backing files during reflective loading? This would allow attackers to better conceal the traces of reflective loading.

Based on Patrick Wardle’s implementation, we developed a reflective loader capable of assigning arbitrary file paths as backing files. Implementation details are provided in Appendix D. Using this loader, directories such as FMCore.framework can be specified as backing files (Figure 4.2).

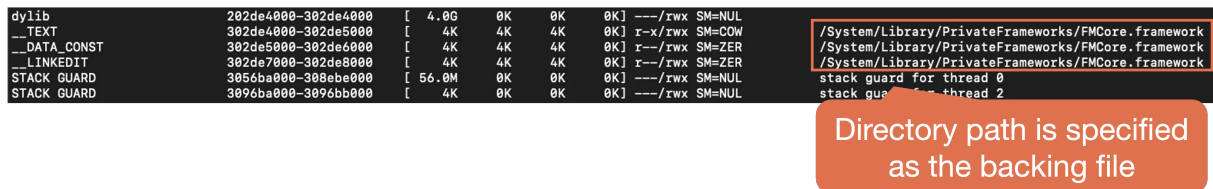


Figure 4.2: This figure shows `vmmap` output of reflective loader with the `FMCore.framework` directory specified as backing file.

Even if this hypothesis is correct, one question remains. If arbitrary file paths can be specified as backing files, why did the attackers choose directories and symlinks? Specifying unused dylibs as backing files should be stealthier. While no definitive answer is available at present, this question may be resolved in the future through analysis of macOS implant samples, should they become available.

4.6 XProtectRemediatorCardboardCutout

XProtectRemediatorCardboardCutout was added in XPR version 122 (January 2024). It is implemented using the RemediationBuilder DSL, with remediation logic defined as shown in Listing 4.11:

Listing 4.11 This listing shows pseudo Swift code that defines the remediation logic for the CardboardCutout scanner.

```
CardboardCutoutRemediator {
    Service(tag: nil) {
        ExecutableIsUntrusted(true) // (1)
        ExecutableRevoked(true) // (2)
    }
}
```

This scanner detects and removes launchd services that meet the following two conditions:

- Executable files of registered launchd services are not trusted (Listing 4.11 (1)).
- Notarization tickets of executable files of registered launchd services have been revoked (Listing 4.11 (2)).

The first condition corresponds to either of the following: the executable's code identifiers cannot be obtained, or the code certificate arrays are null. This information is retrieved using the `SecStaticCodeCreateWithPathAndAttributes` and `SecCodeCopySigningInformation` functions.

The second condition is verified using the `SecAssessmentTicketLookup` function. For details on methods to check whether notarization tickets have been revoked, see Patrick Wardle's book [68].

Both of two conditions can typically be satisfied by registering a malware sample with a revoked notarization ticket as a launch agent or launch daemon. For this reason, the CardboardCutout

scanner is considered generic rather than targeting a specific malware family.

4.7 XProtectRemediatorSheepSwap

XProtectRemediatorSheepSwap has been included since the initial public release of XPR. It is reported to remove Adload [12]. Analysis of the YARA rules in the SheepSwap scanner suggests that the malware dropped by Shlayer is also targeted [53].

Listing 4.12 represents the remediation/detection logic when the scanner runs under a logged-in user account. When run as root, it applies different logic, the details of which remain unknown.

Listing 4.12 This listing shows pseudo Swift code that defines the remediation logic for the SheepSwap scanner.

```
let sheepSwapYara1 = ""
rule sheepswap_variant_1 {
// .. snipped for brevity
""

let sheepSwapYara2 = ""
rule hunt_sheepswap_extension_obfuscation { // (3)
// .. snipped for brevity
""

let sheepSwapYara3 = ""
private rule JavaScript {
// .. snipped for brevity
""

SheepSwapRemediator {
    // For remediation (1)
    SafariAppExtension {
        ExtensionBinaryYara(YaraMatcher(sheepSwapYara1))
    }
    SafariAppExtension {
        ExtensionBinaryYara(YaraMatcher(sheepSwapYara2))
        JavaScriptYara(YaraMatcher(sheepSwapYara3))
    }
    // For detection (2)
    SafariAppExtension {
        JavaScriptYara(YaraMatcher(sheepSwapYara3))
    }.reportOnly()
    SafariAppExtension {
        ExtensionBinaryYara(YaraMatcher(sheepSwapYara2))
    }.reportOnly()
}
```

A distinctive feature of this scanner is that it targets Safari app extensions for removal (Listing 4.12). Among the scanners in XPR version 145, the SheepSwap scanner is the only one

that uses the RemediationBuilder DSL to define logic for removing Safari app extensions.

To remediate Safari app extensions, it is first necessary to enumerate them. This enumeration is performed via the `LSApplicationExtensionRecord` class in the Core Services framework. Using this class, the system-installed app extensions can be enumerated, and Safari app extensions are identified from the results. Listing 4.13 shows code that enumerates app extensions using the `LSApplicationExtensionRecord` class.

Listing 4.13 This listing shows code that enumerates app extensions using the `LSApplicationExtensionRecord` class.

```
id enumerator = [LSApplicationExtensionRecord enumeratorWithOptions:0];
for (id element in enumerator) {
    NSLog(@"Element: %@", element);
}
```

When scanning Safari app extensions, two conditions can be applied: `ExtensionBinaryYara` and `JavaScriptYara`. `ExtensionBinaryYara` checks whether YARA rules match the executable file of a Safari app extension, while `JavaScriptYara` checks whether YARA rules match the JavaScript code within them. JavaScript scanning is performed by enumerating `.js` files in a Safari app extension bundle and checking them against the YARA rules.

This scanner also includes both remediation and detection logic (Listing 4.12 (1) and (2)). The detection logic makes use of YARA rules prefixed with `hunt_` (Listing 4.12 (3)). Based on the naming, these `hunt_` rules appear to serve a threat-hunting purpose and may have been deployed by Apple to track emerging threats. Such rules are found not only in the SheepSwap scanner but also in other scanners, as discussed in Section 4.8.

4.8 Findings Summary & Implications

Finally, we summarize the findings revealed in the preceding sections and discuss their implications. The analysis of XPR scanners highlights the following three findings.

Finding (1): XPR analysis can reveal characteristics of malware not documented in public sources. In some cases, this includes information based on Apple-exclusive threat intelligence.

Examples include the RoachFlight scanner (Section 4.4) and the RedPine scanner (Section 4.5). In the case of the RoachFlight scanner, analysis revealed the previously unknown Smooth Operator sample. In the RedPine scanner, characteristics of TriangleDB macOS implant samples were identified. Even for well-known malware such as Adload, new behaviors not documented publicly were revealed, as discussed in Section 4.1.2. This type of information is based on Apple-exclusive threat intelligence and is particularly valuable.

Finding (2): Individual XPR scanners do not always correspond one-to-one with specific malware families.

Examples include the RoachFlight scanner (Section 4.4) and the RankStank scanner (Section

4.3), both of which target Smooth Operator. Additionally, scanners such as the BadGacha scanner (Section 4.2) and the CardboardCutout scanner (Section 4.6) are not tied to specific malware families, since their generic detection logic is intended to cover multiple families.

Finding (3): XPR scanners are not necessarily designed solely for remediation; some are dedicated to detection, and others serve both remediation and detection purposes.

Examples include scanners that focus exclusively on detection, such as the RedPine scanner (Section 4.5) and the BadGacha scanner (Section 4.2). These scanners perform detection only and do not remediate threats. Furthermore, the SheepSwap scanner includes logic that appear to serve threat-hunting purposes. Similar logic is also found in other scanners, including the WaterNet scanner, the Pirrit scanner, and the Bundlore scanner [45]. A number of YARA rules beginning with `hunt_` exist, indicating that threat-hunting logic is embedded within several scanners.¹⁷

Implications based on Findings (1)–(3) are as follows:

Implication from Finding (1): Finding (1) highlights the importance of regularly analyzing XPR scanners. This implication is particularly relevant for threat researchers, as such analysis can provide insights into Apple-exclusive threat intelligence. Although analyzing stripped Swift binaries presents many challenges, the techniques and tools described in Chapter 2 enable effective analysis.

Implication from Finding (2): Finding (2) suggests that Apple may name XPR scanners not only by malware families but also by other categories, such as detection methods. For example, BadGacha and CardboardCutout do not correspond to specific malware families but instead to methods for detecting Gatekeeper sidestepping attempts, processes without backing files, or launchd services with revoked notarization tickets. However, since only a limited number of cases support this observation, no definitive conclusion can yet be drawn. Careful evaluation based on future XPR updates will be necessary.

Implications from Finding (3): Finding (3) indicates that Apple may use XPR not solely for remediation but also for discovering new threats. This view is consistent with XPR’s collection of provenance extended attributes. This mechanism seems designed to provide Apple with intelligence on previously unknown malware, as discussed earlier in Section 3.5. Another implication is that XPR detection results could be integrated into EDR products to support threat discovery, making this finding relevant for Blue Teams and EDR product developers. Furthermore, since detection does not imply remediation, cases where threats are flagged may require separate, detailed investigation depending on circumstances.

¹⁷The YARA rules used by each XPR scanner are published in [37]. By searching for the string `hunt_` in this repository, it is possible to confirm which other scanners include YARA rules with this prefix.

Chapter 5

Vulnerability Research

This chapter describes vulnerabilities found in XPR. Note that all vulnerabilities discussed here were reported via the Apple Bug Bounty Program and disclosed following responsible disclosure. After Apple released fixes, we published these findings as part of this research.

5.1 CVE-2024-23201 (“Lame” Denial-of-Service Vulnerability)

CVE-2024-23201 was a Denial-of-Service (DoS) vulnerability in XPR. The vulnerability allowed an attacker to forcibly terminate XPR processes using `pkill`, thereby disabling remediation. Listing 5.1 shows a proof-of-concept.

Listing 5.1 This listing shows exploit code for CVE-2024-23201

```
while true; do pkill XProtect 2>/dev/null; done
```

Apple mitigated the issue by granting the `com.apple.private.security.signal-exempt` entitlement to XPR binaries. This entitlement prevents XPR processes from being forcibly terminated in response to `SIGTERM` signals. The final fix was implemented in `libxpc` and shipped with macOS Sonoma 14.3. The details of this fix are unknown.

5.2 CVE-2024-40843 (TCC bypass)

CVE-2024-40843 was a TCC bypass vulnerability in XPR. Proof-of-concept code is published on GitHub [69].

A related issue was reported at Black Hat EU 2022 for Windows EDR products [70]. That issue was a classic TOCTOU vulnerability in which a file could be replaced with a junction to a different file between detection and deletion, enabling deletion with administrator privileges. CVE-2024-40843 can be seen as the macOS analogue: by replacing a detected file with a different file via a symbolic link immediately before deletion, an attacker could delete files.

File deletion on macOS differs from Windows and, in some cases, is subject to TCC restrictions. For example, deleting files in the `Desktop` or `Documents` folders may require explicit user consent. Some XPR scanners hold entitlements that grant Full Disk Access, so a vulnerability that enables file deletion via such a scanner results in a TCC bypass.

Although any scanner with a Full Disk Access entitlement could be abused, we focused on `XProtectRemediatorBlueTop`. The BlueTop scanner runs as root while its remediation targets are plist files under `~/Library/LaunchAgents` that are writable by non-privileged users. Thus, exploiting the BlueTop scanner can lead not only to TCC bypass but also to privilege escalation from users to root.

The steps to exploit the vulnerability were as follows:

- (1) Create a pseudo-malware binary that matches one of the YARA rules in the BlueTop scanner
- (2) Place a plist file under `~/Library/LaunchAgents` to register the pseudo-malware with `launchd`
- (3) Wait for a periodic scan of the BlueTop scanner running as root
- (4) Monitor the Unified Log to detect when the YARA rule matches
- (5) At the confirmed timing, replace the pseudo-malware with the intended target file using a symbolic link
- (6) Have the scanner delete the target file with Full Disk Access and root privileges

The timing in step (4) can be observed via `log stream`. When the YARA rule matches, an entry like the following is emitted: “Rule YaraRule[macos_bluego]: Hit” (Listing 5.2).

Listing 5.2 This listing shows an example log entry when the YARA rule matches

```
Default    0x0    3670    0    XProtectRemediatorBlueTop:
↳ [com.apple.XProtectFramework.PluginAPI:YARARuleMatchV4] Rule
↳ YaraRule[macos_bluego]: Hit
```

To actually exploit this vulnerability, an attacker must craft a pseudo-malware sample that matches the YARA rules and thereby trigger XPR detection. Note that some third-party security products collect XPR detection results as telemetry; hence attacks that deliberately trigger XPR detections are noisy and have poor stealth against endpoints running such products. However, on consumer devices, detection by XPR typically does not present an obvious notification to the user, so creating pseudo-malware to induce detection may not appear suspicious to a typical user.

Exploitation via the BlueTop scanner is more likely to be noticed: Background Task Management will surface a notification to the user, increasing the chance of user awareness. That said, Patrick Wardle outlined several techniques for evading this notification in his DEF CON presentation [71]. By combining such techniques, it may still be possible to exploit the vulnerability while minimizing user-visible indicators.

Chapter 6

Conclusion

This paper presented the results of reverse engineering XPR together with the analysis methodology. Our analysis revealed that XPR employs a dedicated DSL to concisely define remediation/detection logic. By examining the logic used in each scanner, we showed that some XPR scanners include information based on Apple-exclusive threat intelligence. We also demonstrated that XPR is not used solely for remediation: in certain cases, it only performs threat detection. Finally, we introduced vulnerability research on XPR and showed that elementary issues similar to those found in other platforms' security products also exist in XPR.

These findings provide a detailed view of XPR's internals and vulnerabilities, and lead to several implications. For threat researchers, XPR analysis results offer a potential source of insights into Apple-exclusive threat intelligence. For blue teams, our results highlight the importance of investigating XPR detection events more closely, even when they appear as remediation outcomes because some scanners only perform detection. For red teams, we demonstrated the possibility of discovering new XPR vulnerabilities by drawing parallels with bugs previously identified in EDR and AV products on other platforms.

As the macOS user base continues to grow, attackers will persist in targeting the platform. Security mechanisms such as Gatekeeper and Notarization remain strong defenses, yet threats that bypass them continue to appear. XPR's role as a standard security mechanism will therefore become increasingly important, with the number of scanners likely to expand. We hope the results of this study will support future research and aid investigations into newly added scanners.

Acknowledgments

We used AI-based tools (OpenAI ChatGPT and Claude) to support English proofreading and to refine the clarity of some passages in this paper. All responsibility for the content remains with the authors.

We would like to express our sincere gratitude to Howard Oakley and Phil Stokes for kindly reading through an early draft of this paper and providing valuable comments.

Appendix A

Vnode Rapid Aging

Vnode Rapid Aging is a feature that suppresses access-time (atime) updates and can be toggled on a per-process basis. A process can enable or disable the feature by calling the `sysctl` function as shown below. No special entitlements are required to enable it. Listing A.1 shows sample code for enabling Vnode Rapid Aging.

Listing A.1 This listing shows code to enable Vnode Rapid Aging.

```
void enable_vnode_rapid_aging(int enabled) {
    int mib[] = {CTL_KERN, KERN_RAGEVNODE};
    if (sysctl(mib, 2, NULL, NULL, &enabled, sizeof(enabled)) != 0) {
        perror("Failed to call sysctl");
    }
}
```

Although this feature is not described in Apple's official documentation, it was discussed on the darwin-dev mailing list in 2012 [72]. Comments about the feature also exist in the XNU HFS source code, where atime updates are skipped when Vnode Rapid Aging is enabled [73] (Listing A.2). This behavior is not limited to HFS; it is also implemented in APFS.

Listing A.2 This listing shows an excerpt from the HFS source code (hfs_cnode.c). The `vnode_israge` function determines whether Vnode Rapid Aging is enabled.

```
/*
 * Skip access time updates if:
 *     . MNT_NOATIME is set
 *     . a file system freeze is in progress
 *     . a file system resize is in progress
 *     . the vnode associated with this cnode is marked for rapid
↪ aging
 */
if (cp->c_touch_acctime) {
    if ((vfs_flags(hfsmp->hfs_mp) & MNT_NOATIME) ||
        (hfsmp->hfs_freezing_proc != NULL) ||
        (hfsmp->hfs_flags & HFS_RESIZE_IN_PROGRESS) ||
        (cp->c_vp && vnode_israge(cp->c_vp)))
        cp->c_touch_acctime = FALSE;
}
```

Appendix B

Provenance Sandbox

The Provenance Sandbox is a mechanism that enables tracking of which application has interacted with a given file. A special extended attribute named `com.apple.provenance` is attached to the target application and registered in the ExecPolicy database via `syspolicyd`. The target process is then run in the Provenance Sandbox through `Quarantine.kext`. A process running in this sandbox inherits and propagates the provenance attribute when creating or modifying files.

This mechanism has been introduced in macOS Ventura and was first reported by Howard Oakley [74]. As explained earlier, XPR leverages this mechanism to track the origins of files subject to remediation/detection. The Apple Platform Security Guide also suggests that systems other than XPR use this mechanism [6]. For example, in its description of XProtect Behavior Service, it states that information detected by the service is ultimately sent to Apple together with details of the software responsible for downloading it:

In addition, XProtect contains an advanced engine to detect unknown malware based on behavioral analysis. Information about malware detected by this engine, including what software was ultimately responsible for downloading it, is used to improve XProtect signatures and macOS security.

This section explains how the Provenance Sandbox is enabled, its behaviors, how third parties can make use of the mechanism, and finally, vulnerabilities that allow it to be bypassed.

It should be noted that despite the name Provenance *Sandbox*, the mechanism differs from the common use of the term sandbox in computer security. Unlike the App Sandbox, its purpose is not to minimize the impact if the process is compromised. However, there are certain similarities with the App Sandbox, which is likely why the term sandbox was adopted. These similarities are discussed in detail later.

B.1 How the Provenance Sandbox Is Enabled

Figure B.1 shows a sequence diagram illustrating how the Provenance Sandbox is enabled. This flow represents what happens when an application is downloaded and executed for the

first time on a system.

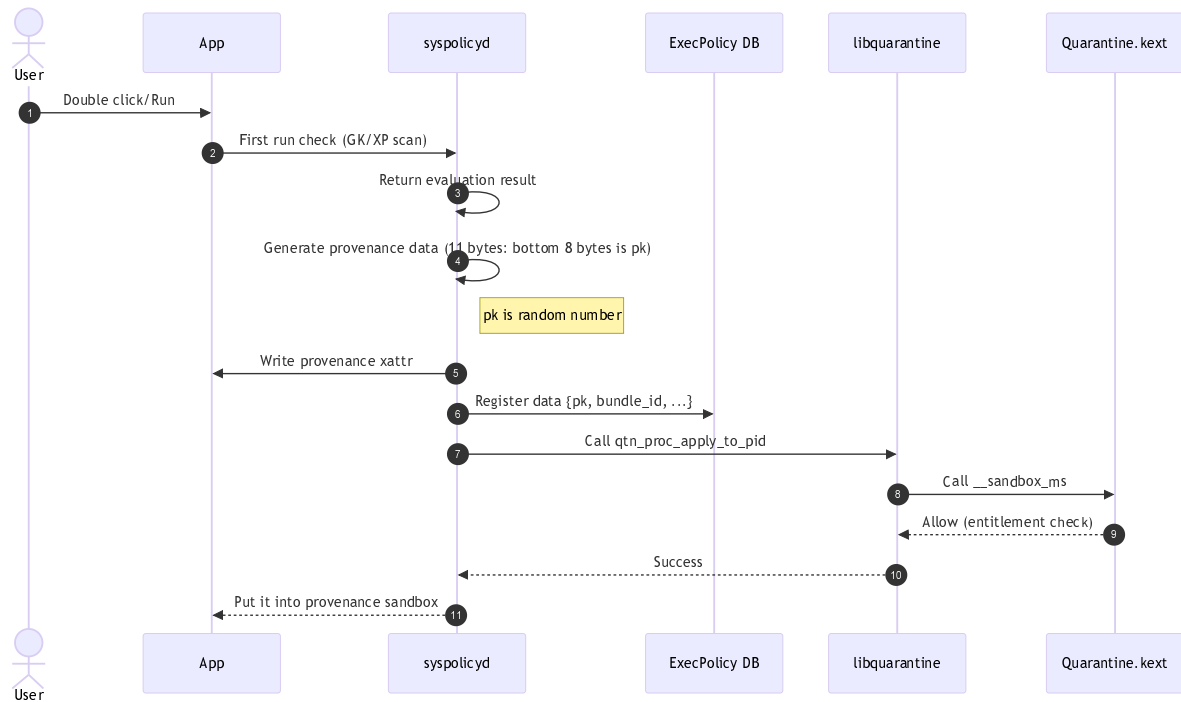


Figure B.1: This sequence diagram illustrates how the Provenance Sandbox is enabled. Here GK represents Gatekeeper, XP represents XProtect, and pk represents primary key.

The process of running an application within the Provenance Sandbox can be broken down into the following steps. The numbers below correspond to the steps in Figure B.1.

- (1)–(3) When an application is executed, Gatekeeper and XProtect scans are performed.
- (4) `syspolicyd` generates an 11-byte integer value.
- (5) This 11-byte value is written as an extended attribute named `com.apple.provenance` to the application bundle. The value is split into the upper 3 bytes and the lower 8 bytes. The lower 8 bytes are random numbers generated by `arc4random` and serve as the primary key in the `provenance_tracking` table of the `ExecPolicy` database. The role of the upper 3 bytes remains unknown.
- (6) `syspolicyd` registers this data in the `provenance_tracking` table of the `ExecPolicy` database. The stored data includes the application's bundle ID, code signature information, and more. The database's primary key is the random value generated in step (4).
- (7)–(11) Finally, `syspolicyd` calls `qtn_proc_apply_to_pid` and runs the process in the Provenance Sandbox via `Quarantine.kext`.

At step (5), the provenance extended attribute is written to the application bundle and persists on disk. On subsequent executions, the application that already has this extended attribute is run in the Provenance Sandbox directly through steps (7)–(11). Although the example here focuses on an application bundle, the same behavior applies to any dylib or executable file with the provenance extended attribute.

To illustrate steps (1)–(6), Listing B.1 shows a portion of `syspolicyd` logs captured from the

Unified Log entries when running *Visual Studio Code*. After the “GK evaluateResult” log entry, we can see the messages “Created provenance data for target” and “Wrote provenance data on target,” showing that provenance data was generated and written to the application.

Listing B.1 This listing shows an excerpt of syspolicyd logs from the Unified Log entries when running *Visual Studio Code*.

```
Info          0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] GK Xprotect results: PST: (path:
↳ 2a7545b632d3156f), (team: UBF8T346G9), (id: (null)), (bundle_id: (null)),
↳ XPScan: 0,-8084322555107053820,2025-01-13 13:08:13
↳ +0000,(null),(null),file:///Applications/Visual%20Studio%20Code.app/
... (snipped for brevity)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] GK evaluateScanResult: 0, PST: (path:
↳ 2a7545b632d3156f), (team: UBF8T346G9), (id: com.microsoft.VSCode),
↳ (bundle_id: com.microsoft.VSCode), 1, 0, 1, 0, 4, 4, 0
... (snipped for brevity)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Created provenance data for target:
↳ TA(25df15983a02ace9, 2), PST: (path: 2a7545b632d3156f), (team: UBF8T346G9),
↳ (id: com.microsoft.VSCode), (bundle_id: com.microsoft.VSCode)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Handling provenance root:
↳ TA(25df15983a02ace9, 2)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Wrote provenance data on target:
↳ TA(25df15983a02ace9, 2), PST: (path: 2a7545b632d3156f), (team: UBF8T346G9),
↳ (id: com.microsoft.VSCode), (bundle_id: com.microsoft.VSCode)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Putting executable into provenance with
↳ metadata: TA(25df15983a02ace9, 2)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Putting process into provenance tracking
↳ with metadata: 732, TA(25df15983a02ace9, 2)
Default      0x0          196    0    syspolicyd:
↳ [com.apple.syspolicy.exec:default] Tracking process with attributes: 732,
↳ TA(25df15983a02ace9, 2)
```

Listing B.2 shows the contents of the provenance extended attribute for the *Visual Studio Code* application bundle. The lower 8 bytes of this 11-byte value are used as the primary key in the provenance_tracking table of the ExecPolicy database.

Next, let’s examine the data registered in the provenance_tracking table of the ExecPolicy database. Listing B.3 shows the table contents after running *Visual Studio Code* in the same example. The value 2728923642762276073 (0x25df15983a02ace9 in hexadecimal) appears in the pk column. When interpreted in little-endian format, this value matches the lower 8 bytes shown in Listing B.2.

Next, we explain the details of steps (7)–(11). The APIs used for assigning a pro-

Listing B.2 This listing shows the contents of the provenance extended attribute for the *Visual Studio Code* application bundle.

```
% xattr -px com.apple.provenance /Applications/Visual\ Studio\ Code.app
01 02 00 E9 AC 02 3A 98 15 DF 25
```

Listing B.3 This listing shows the contents of the `provenance_tracking` table in the `ExecPolicy` database after running *Visual Studio Code*.

```
sqlite> .dump provenance_tracking
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE provenance_tracking ( pk INTEGER PRIMARY KEY, url TEXT NOT NULL,
↪ bundle_id TEXT, cdhash TEXT, team_identifier TEXT, signing_identifier
↪ TEXT, flags INTEGER, timestamp INTEGER NOT NULL, link_pk INTEGER);
INSERT INTO provenance_tracking
↪ VALUES(2728923642762276073,'/Applications/Visual Studio
↪ Code.app','com.microsoft.VSCode','c770be0fce7103c64ccdecc7069d8ac363759d8d
↪ ','UBF8T346G9','com.microsoft.VSCode',2,1736773699,0);
```

cess to the Provenance Sandbox are functions prefixed with `qtn_` and provided by `/usr/lib/system/libquarantine.dylib`. Typical API calls used for this purpose are shown in Listing B.4.

Here, when the `qtn_proc_apply_to_pid` function is called, `Quarantine.kext` checks for the presence of either the `com.apple.private.quarantine.control` or `com.apple.private.quarantine.control-add` entitlement. If neither entitlement is present, the process is not allowed to run in the Provenance Sandbox. This check is intended to prevent attackers from reusing a provenance extended attribute from another application to launch a process in the Provenance Sandbox.

B.2 Behaviors of the Provenance Sandbox

When the Provenance Sandbox is enabled for a running process, it exhibits the following behaviors:

- (1) During certain file operations¹, the same provenance extended attribute attached to an application is automatically attached to the files.
- (2) Child processes are automatically executed within the same Provenance Sandbox.

These behaviors resemble those of the App Sandbox. In the App Sandbox, a quarantine extended attribute is attached to files created by a sandboxed processes, and child processes are automatically run in the same sandbox. The Provenance Sandbox can be understood as a mechanism that substitutes the provenance extended attribute in place of the quarantine extended attribute in the App Sandbox behaviors.

¹The affected file operations are 11 types: `create`, `deleteextattr`, `open` (with write mode), `setacl`, `setattrlist`, `setextattr`, `setflags`, `setmode`, `setowner`, `setutimes`, `truncate`, `link`, and `rename` [75].

Listing B.4 This listing shows typical API calls used until a process is assigned to the Provenance Sandbox.

```
// Allocate a qtn_proc_t structure
qtn_proc_t* qpt = qtn_proc_alloc();

// Initialize the structure with data for the given PID
qtn_proc_init_with_pid(qpt, pid);

// Assign provenance tracking data to the qtn_proc_t structure
qtn_proc_set_tracking_data(qpt, [provenanceData bytes], [provenanceData
↪ length]);

// Update the flags field, OR-ing with 0x200 to mark the process for Provenance
↪ Sandbox assignment
qtn_proc_set_flags(qpt, qtn_proc_get_flags(qpt) | 0x200);

// Invoke Quarantine.kext with __sandbox_ms to deploy the process into the
↪ Provenance Sandbox
qtn_proc_apply_to_pid(qpt, pid);
```

B.3 How Dropped Executables and dylibs Are Tracked in the ExecPolicy Database

When an application running in the Provenance Sandbox drops an executable or dylib and then executes it, an interesting behavior can be observed. Specifically, when the executable is run or the dylib is loaded, information about that executable or dylib is recorded in the `provenance_tracking` table of the ExecPolicy database.

This can be illustrated with a concrete example. Consider the following case²:

- Attach the provenance extended attribute of Google Chrome to `b.dylib` (reproducing a situation in which `b.dylib` is dropped in Google Chrome's Provenance Sandbox)
- Have `a.out`, which does not have a provenance extended attribute, load `b.dylib`

In this example, the `provenance_tracking` table contains a record showing that `b.dylib` was loaded (Listing B.5 (1)). Here, a new primary key (-8402853012194506118) is created for `b.dylib` at (1). In addition, the record for `b.dylib` includes a `link_pk` (5957287531021454401), which corresponds to the primary key of *Google Chrome* at (2).

This behavior can be leveraged to trace the execution of malware that drops and runs multiple payloads. Furthermore, as discussed later, it can also be used to trace the execution of reflective loaders that utilize the `NSLink` and `NSCreateImageFromMemory` APIs.

²The code used for this demonstration is available on GitHub [75].

Listing B.5 This listing shows the contents of the `provenance_tracking` table in the ExecPolicy database after running *Google Chrome*.

```
CREATE TABLE provenance_tracking ( pk INTEGER PRIMARY KEY, url TEXT NOT NULL,  
↪ bundle_id TEXT, cdhash TEXT, team_identifier TEXT, signing_identifier  
↪ TEXT, flags INTEGER, timestamp INTEGER NOT NULL, link_pk INTEGER);  
... (snipped for brevity)  
INSERT INTO provenance_tracking  
↪ VALUES(-8402853012194506118, '/Users/user/ProvenanceChecker/demo/b.dylib', 'J  
↪ NOT_A_BUNDLE', NULL, NULL, NULL, 2, 1758287022, 5957287531021454401); (1)  
INSERT INTO provenance_tracking  
↪ VALUES(5957287531021454401, '/Applications/Google  
↪ Chrome.app', 'com.google.Chrome', '7a07db0c7eb722aebc7b64922264b3a9440c22f4'  
↪ , 'EQHXZ8M8AV', 'com.google.Chrome', 2, 1755008034, 0); (2)  
... (snipped for brevity)
```

B.4 Usage Examples of the Provenance Sandbox

In Section 3.5, we explained how XPR leverages provenance extended attributes automatically attached by the Provenance Sandbox. To demonstrate how this mechanism can be used by third parties, we published a minimal proof-of-concept implementation *ShowProvenanceInfo* in a GitHub repository [76]. This example retrieves provenance extended attributes from files, looks up the corresponding application information in the `provenance_tracking` database, and outputs the results. It provides a lightweight reference for researchers and incident responders who wish to experiment with provenance data. Building on this simple demonstration, the following subsections present more practical usage scenarios where provenance information can support compromise assessment and malware analysis.

Identifying the Scope of Compromise

When it is known that attacker-controlled applications were executed in the Provenance Sandbox, compromised files can be identified by examining their provenance extended attribute values. As a practical implementation, we added support for collecting provenance extended attributes to *Aftermath*, a macOS incident response framework. This integration enables investigators to correlate provenance data with other forensic artifacts during an incident assessment.

Identifying an Application That Achieved Persistence

Another use case is identifying an application that achieves persistence. By inspecting the provenance extended attribute of a plist file in `~/Library/LaunchAgents`, it is possible to determine which application created that plist file and thereby established persistence.

Identifying an Application Using a Reflective Loader

The Provenance Sandbox can also help identify an application that uses a reflective loader implemented with `NSCreateObjectFileImageFromMemory` and `NSLinkModule`. In dyld3, the re-

fective loader implementation using these APIs has been modified to save a dylib to disk once and then load it via `dlopen` [67]. As a result, when such an application is executed in the Provenance Sandbox, a provenance extended attribute is attached to the dylib written to disk. A subsequent `dlopen` call involving that dylib is then recorded in the `provenance_tracking` table of the `ExecPolicy` database. By inspecting this table, it becomes possible to identify which application uses the reflective loader.

B.5 How to Bypass the Provenance Sandbox

Finally, we briefly introduce Provenance Sandbox bypass vulnerabilities. Note that these vulnerabilities have been fixed in updates to macOS Sequoia 15.

Approaches to discovering Provenance Sandbox bypasses can be grouped into two categories:

- (1) Discover methods for creating files without having provenance extended attributes attached by processes running in the Provenance Sandbox
- (2) Discover methods for executing processes in ways that do not cause them to be assigned to the Provenance Sandbox

Exploiting the `uchg` File Flag

One example in category (1) is using the `uchg` (user immutable) file flag to prevent provenance extended attributes from being attached to created files. This approach was inspired by a Gatekeeper bypass technique presented at CODE BLUE 2023 [77]. Specifically, if an application bundle has the `uchg` flag set, provenance extended attribute is not written to it, which breaks the persistence of the attribute. As a result, subsequent executions of the same application will not cause it to be assigned to the Provenance Sandbox.

Executing Processes in Ways That Avoid the Provenance Sandbox

One example in category (2) is executing code without triggering assignment to the Provenance Sandbox, even when the file has a provenance extended attribute. A common case is launching a file via Launch Services—for example, using the `open` command. If an attacker drops a `.terminal` file and launches it with `open`, the process is executed outside the Provenance Sandbox. Another case involves Automator: dropping an Automator workflow bundle and executing it with the `automator` command can result in a workflow running outside the Provenance Sandbox.

Appendix C

Is XPR Truly Using Swift’s Result Builders?

Result builders allow the creation of DSLs by applying the `@resultBuilder` attribute to structs or enums and defining methods such as `buildBlock` and `buildExpression`. In Section 3.4, we explained that enums like `FileRemediationBuilder` are defined to implement the Remediation-Builder DSL. However, since symbols are stripped, we cannot directly confirm whether methods specific to result builders are present in these enums. Moreover, attribute information applied to enums is lost during compilation, so it is also unclear whether the `@resultBuilder` attribute itself is applied to enums. Thus, verifying that XPR uses result builders is not straightforward.

To address this, we used a “verification through reimplementation” approach. Specifically, we reimplemented part of the `RemediationBuilder` DSL and functionality equivalent to `XProtectRemediatorEicar` using it. We then obtained decompilation results from both implementations and compared them. The reimplementation is published in a repository called *OpenRemediationBuilder* [78].

Next, we explain the comparison method. In this analysis, we compared decompilation results at a location where code corresponding to `RemediationArrayBuilder` DSL in Listing 3.8 (2) is executed. This code is automatically generated by the Swift compiler based on methods defined in `RemediationArrayBuilder`. Therefore, if built with the same version compiler as the Swift version that compiled the `XProtectRemediatorEicar` scanner, almost the same code should be generated, and decompilation results should be almost identical.

Figure Figure C.1 shows a comparison of *Binary Ninja* HLIL output. We observed that Swift runtime function call patterns (such as `_swift_initStackObject` and `_swift_allocObject`) matched. In addition, function call patterns beyond runtime functions (e.g., calls to `buildBlock`) and access patterns to struct members allocated on the stack also matched. From these observations, we conclude that XPR does indeed use result builders.

HLIL of OpenRemediationBuilder

```

100004e20  int128_t (*)[] sub_100004e20()
100004e4b      int128_t var_68
100004e4b      sub_10000770('/tmp/eic', 'ar\x00\x00\x00\x00\x00\x00\x00\x00', &var_68)
100004e66      void var_b0
100004e66      void* rax_2 = _swift_initStackObject(sub_100002cc0(&data_1000124a0), &var_b0)
100004e75      *(rax_2 + 0x10) = data_10000b360
100004e80      *(rax_2 + 0x38) = &type metadata for OpenRemediationBuilder.File
100004e8b      *(rax_2 + 0x40) = &pwt of OpenRemediationBuilder...diationBuilder.RemediationConvertible
100004ea8      void* rax_3 = _swift_allocObject(&data_1000106b8, 0x58, 7)
100004ea8      *(rax_2 + 0x20) = rax_3
100004eb9      *(rax_3 + 0x10) = var_68
100004ebd      int128_t var_58
100004ebd      *(rax_3 + 0x20) = var_58
100004ec1      int128_t var_48
100004ec1      *(rax_3 + 0x30) = var_48
100004ec5      int128_t var_38
100004ec5      *(rax_3 + 0x40) = var_38
100004ecd      int64_t var_28
100004ecd      *(rax_3 + 0x50) = var_28
100004ed4      sub_100005020(&var_68)
100004edc      int64_t (* result)() = sub_100007990(rax_2)
100004ee7      sub_100005080(&var_68)
100004ee7      _swift_release(rax_2)
100004ef4      return result

```

HLIL of XProtectRemediatorEicar

```

100097198  int128_t (*)[] sub_100097198()
1000971c4      int128_t var_68
1000971c4      sub_100009700('/tmp/eic', 'ar\x00\x00\x00\x00\x00\x00\x00\x00', sub_100097280, &var_68)
1000971df      void var_b0
1000971df      void* rax_2 = _swift_initStackObject(sub_100097bb0(&data_1000f2ad0), &var_b0)
1000971f9      *(rax_2 + 0x10) = data_1000b1f10
1000971f9      *(rax_2 + 0x38) = &type metadata for RemediationBuilder.File
100097204      *(rax_2 + 0x40) = &pwt of RemediationBuilder...diationBuilder.RemediationConvertible
100097219      void* rax_3 = _swift_allocObject(&data_1000dede0, 0x58, 7)
10009721e      *(rax_2 + 0x20) = rax_3
100097232      *(rax_3 + 0x10) = var_68
100097236      int128_t var_58
100097236      *(rax_3 + 0x20) = var_58
10009723a      int128_t var_48
10009723a      *(rax_3 + 0x30) = var_48
10009723e      int128_t var_38
10009723e      *(rax_3 + 0x40) = var_38
100097246      int64_t var_28
100097246      *(rax_3 + 0x50) = var_28
10009724d      sub_100097c30(&var_68)
100097255      int64_t (* result)() = sub_100007e90(rax_2)
100097260      sub_100097c90(&var_68)
100097268      _swift_release(rax_2)
10009727d      return result

```

Figure C.1: This figure shows a comparison of *Binary Ninja* HLIL output for the OpenRemediationBuilder reimplementa-tion (left) and the Eicar scanner (right).

Appendix D

Custom Reflective Loader Implementation Details

Here, we describe the implementation details of a reflective loader that allows an arbitrary file path to be specified as a backing file. Before going into implementation details, we first explain how the file path information specified in `HasLoadedLibrary` (introduced in Section 4.5.2) is actually used for detection.

Whether the library specified in `HasLoadedLibrary` is loaded in a target process is determined using the `VMUProcessDescription` class from the Symbolication framework. This class provides a method to obtain the `task_dyld_info` struct from a target process via the `task_info` function. The `task_dyld_info` struct includes the `all_image_info_addr` member (Listing D.1 (1)), which points to the `dyld_all_image_infos` struct. This struct contains metadata about all images loaded by the target process.

Listing D.1 This listing shows the `task_dyld_info` struct definition (excerpt from `task_info.h` header)

```
struct task_dyld_info {
    mach_vm_address_t    all_image_info_addr; // (1)
    mach_vm_size_t       all_image_info_size;
    integer_t            all_image_info_format;
};
```

The `infoArray` member of the `dyld_all_image_infos` struct contains a pointer to an array of `dyld_image_info` structs (Listing D.2 (1)). Each `dyld_image_info` struct includes the `imageFilePath` member, which stores the file path of a loaded dylib (Listing D.2 (2)). By referencing this `imageFilePath` member, the file paths of the backing files for currently loaded dylibs can be obtained. These file paths are compared with the parameter specified in `HasLoadedLibrary` to detect whether the specified library is loaded.

To implement reflective loaders that allow file paths to be specified as backing files, the contents of the `infoArray` structure must be modified. This can be achieved through the following steps:

- (1) Following Patrick Wardle's reflective loader implementation [67], load an in-memory dylib as an image (Listing D.3 (1))
- (2) Call the `task_info` function to obtain the address of the `dyld_all_image_infos` struct (Listing D.3 (2))
- (3) Add an entry for the reflectively loaded dylib to the `infoArray` member of the `dyld_all_image_infos` struct, specifying an arbitrary backing file path in the `imageFilePath` member (Listing D.3 (3))

For the complete implementation, please refer to the `TestSuite/RedPine/macho-dyld-in-memory` directory of XPRTestSuite [45].

Listing D.2 This listing shows the excerpt from dyld_images.h header.

```
struct dyld_image_info { // (2)
    const struct mach_header*    imageLoadAddress;    /* base address image is
    ↪ mapped into */
    const char*                  imagePath;           /* path dyld used to load
    ↪ the image */
    uintptr_t                    imageFileModDate;     /* time_t of image file */
                                                    /* if stat().st_mtime of
    ↪ imagePath does not
    ↪ match imageFileModDate,
    ↪ */
                                                    /* then file has been
    ↪ modified since dyld
    ↪ loaded it */
};
//
// ... snipped for brevity
//
// Must be aligned to support atomic updates
// Note sim cannot assume alignment until all host dylds are new enough
#if TARGET_OS_SIMULATOR
struct dyld_all_image_infos
#else
struct __attribute__((aligned(16))) dyld_all_image_infos
#endif
{
    uint32_t                      version;             /* 1 in Mac OS X 10.4 and
    ↪ 10.5 */
    uint32_t                      infoArrayCount;
#if defined(__cplusplus) && (BUILDING_LIBDYLD || BUILDING_DYLD)
    std::atomic<const struct dyld_image_info*>    infoArray;
#else
    const struct dyld_image_info*    infoArray; // (1)
#endif
    // ... snipped for brevity
};
```

Listing D.3 This listing shows the implementation of reflective loader that can specify file paths as backing files.

```
// Based on the implementation of dlopen_from_memory by Patrick Wardle (see:
↳ https://objectivebythesea.org/v7/talks.html#Speaker_1)
extern "C" void* dlopen_from_memory(void* mh, const char* path, int len) {
    // (1)
    auto image = ImageLoaderMachO::instantiateFromMemory(path,
↳ (macho_header*)mh, len, g_linkContext);

    std::vector<const char*> rpaths;
    ImageLoader::RPathChain loaderRPaths(NULL, &rpaths);
    image->link(g_linkContext, true, false, false, loaderRPaths, path);

    ImageLoader::InitializerTimingList initializerTimes[1];
    initializerTimes[0].count = 0;
    image->runInitializers(g_linkContext, initializerTimes[0]);

    dyld_all_image_infos* all_images = get_all_image_infos(); // (2)
    append_dylib_info(all_images, (void*)image->machHeader(), path); // (3)

    return (void*)image->machHeader();
}
```

References

- [1] Red Canary, “Red Canary Threat Detection Report.” Accessed: Sep. 21, 2025. [Online]. Available: https://resource.redcanary.com/rs/003-YRU-314/images/2025ThreatDetectionReport_RedCanary.pdf
- [2] P. Wardle, “Byteing back: detection, dissection and protection against macOS stealers,” in *Virus Bulletin Conference 2024 (VB2024)*, Virus Bulletin, Oct. 2024. Accessed: Sep. 03, 2025. [Online]. Available: <https://www.virusbulletin.com/conference/vb2024/abstracts/byteing-back-detection-dissection-and-protection-against-macos-stealers/>
- [3] Jamf Software, Inc., “Security 360: Annual Trends Report.” Accessed: Sep. 21, 2025. [Online]. Available: https://media.jamf.com/documents/white_papers/security-360-2025-Mac-devices.pdf
- [4] P. Wardle, “Mac-ing Sense of the 3CX Supply Chain Attack: Analysis of the macOS Payloads,” in *Black Hat USA 2023 Briefings*, Informa PLC, Aug. 2023. Accessed: Sep. 03, 2025. [Online]. Available: <https://www.blackhat.com/us-23/briefings/schedule/#mac-ing-sense-of-the-3cx-supply-chain-attack-analysis-of-the-macos-payloads>
- [5] A. Larsen, D. Kelly, J. Pisano, M. Golembiewski, M. Williams, and P. Godvin, “North Korea Leverages SaaS Provider in a Targeted Supply Chain Attack.” Accessed: Sep. 25, 2025. [Online]. Available: <https://cloud.google.com/blog/topics/threat-intelligence/north-korea-supply-chain>
- [6] Apple, Inc., *Apple Platform Security*. 2024. Accessed: Sep. 02, 2025. [Online]. Available: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf
- [7] C. Owens, J. Bradley, and P. Wardle, “All Your Macs Are Belong To Us: The Story of CVE-2021-30657 .” Accessed: Sep. 25, 2025. [Online]. Available: <https://objectivebythesea.org/v4/talks.html#All%20Your%20Macs%20Are%20Belong%20To%20Us>
- [8] P. Wardle, “Gatekeeper Exposed; Come, See, Conquer,” in *Shmoocon 2016*, Jan. 2016. Accessed: Sep. 21, 2025. [Online]. Available: <https://speakerdeck.com/patrickwardle/shmoocon-2016-gatekeeper-exposed-come-see-conquer>
- [9] P. Wardle, “Malware Persistence on OS X Yosemite,” in *RSA Conference 2015*, Apr. 2015. Accessed: Sep. 21, 2025. [Online]. Available: <https://s3.amazonaws.com/s3.synack.com/RSAC+2015+Final.pdf>
- [10] J. B. Or, “Gatekeeper’s Achilles heel: Unearthing a macOS vulnerability.” Accessed: Sep. 21, 2025. [Online]. Available: <https://www.microsoft.com/en-us/security/blog/2022/12/19/gatekeepers-achilles-heel-unearthing-a-macos-vulnerability/>

- [11] A. Schmidt, “The Secrets of XProtectRemediator.” Accessed: Sep. 10, 2025. [Online]. Available: <https://alden.io/posts/secrets-of-xprotect/>
- [12] P. Stokes, “A few more of the missing XProtectRemediator names.” Mar. 26, 2024. Accessed: Sep. 10, 2025. [Online]. Available: <https://x.com/philstokes/status/1838598241597710744>
- [13] H. Oakley, “Why XProtect Remediator scans now take longer.” Accessed: Sep. 10, 2025. [Online]. Available: <https://eclecticlight.co/2025/01/03/why-xprotect-remediator-scans-now-take-longer/>
- [14] Jamf Software, Inc., *Jamf Protect Documentation - Telemetry for macOS Event Categories*. 2025. Accessed: Sep. 11, 2025. [Online]. Available: https://learn.jamf.com/en-US/bundle/jamf-protect-documentation/page/Telemetry_for_macOS_Event_Categories.html
- [15] H. Oakley, “What happens when XProtect Remediator discovers real malware?” Accessed: Sep. 10, 2025. [Online]. Available: <https://eclecticlight.co/2022/12/31/what-happens-when-xprotect-remediator-discovers-real-malware/>
- [16] K. Johnson, “A deep dive into macOS TCC.db.” Accessed: Sep. 10, 2025. [Online]. Available: <https://www.rainforestqa.com/blog/macOS-tcc-db-deep-dive>
- [17] C. Fitzl and W. Regula, “Knockout Win Against TCC - 20+ NEW Ways to Bypass Your MacOS Privacy Mechanisms,” in *Black Hat Europe 2022 Briefings*, Informa PLC, Dec. 2022. Accessed: Oct. 01, 2025. [Online]. Available: <https://www.blackhat.com/eu-22/briefings/schedule/#knockout-win-against-tcc---20-new-ways-to-bypass-your-macos-privacy-mechanisms-29272>
- [18] G. Kalman, “XProtect FDA fileleak exploit.” Accessed: Sep. 10, 2025. [Online]. Available: https://x.com/gergely_kalman/status/1838598241597710744
- [19] S. Ashenbrenner, “dmXProtect: Stop, Drop, Shut Malware Down Before It Opens Up Shop.” Accessed: Sep. 03, 2025. [Online]. Available: <https://www.huntress.com/blog/dmxprotect-stop-drop-shut-malware-down-before-it-opens-up-shop>
- [20] Vector 35 Inc., “Binary Ninja.” Accessed: Sep. 23, 2025. [Online]. Available: <https://binary.ninja>
- [21] LLVM Project, “The LLDB Debugger.” Accessed: Sep. 23, 2025. [Online]. Available: <https://lldb.llvm.org/>
- [22] Apple, Inc., “Swift Programming Language.” Accessed: Sep. 23, 2025. [Online]. Available: <https://github.com/swiftlang/swift>
- [23] K. M. Nakagawa, “binja-swift-analyzer.” Accessed: Sep. 15, 2025. [Online]. Available: <https://github.com/FFRI/binja-swift-analyzer>
- [24] K. M. Nakagawa, “binja-missinglink.” Accessed: Sep. 15, 2025. [Online]. Available: <https://github.com/FFRI/binja-missinglink>
- [25] K. M. Nakagawa, “Some useful LLDB scripts for my macOS debugging.” Accessed: Sep. 15, 2025. [Online]. Available: <https://github.com/kohnakagawa/LLDB>
- [26] blacktop, “Ipsw: iOS/macOS research swiss army knife.” Accessed: Sep. 23, 2025. [Online]. Available: <https://github.com/blacktop/ipsw>

- [27] J. Levin, *Disarming Code: System Programming, Debugging & Reverse Engineering on Linux, Android, Darwin*, First edition. United States: Technogeeks Press, 2025. Available: <https://www.amazon.com/dp/0991055500>
- [28] S. Knight, "Swift Metadata." Accessed: Sep. 13, 2025. [Online]. Available: <https://knight.sc/reverse%20engineering/2019/07/17/swift-metadata.html>
- [29] C. Lopez, "Swift Reversing in 2024 It's not so bad :)," in *Objective by the Sea (OBTS) v7.0*, Dec. 2024. Accessed: Sep. 10, 2025. [Online]. Available: https://objectivebythesea.org/v7/talks/OBTS_v7_cLopez.pdf
- [30] T. Jin, "Swift MemoryLayout." Accessed: Sep. 10, 2025. [Online]. Available: <https://github.com/TannerJin/Swift-MemoryLayout>
- [31] LLVM Project, "Scripting Bridge API." Accessed: Oct. 01, 2025. [Online]. Available: <https://lldb.llvm.org/resources/sbapi.html>
- [32] Google LLC., "BinDiff." Accessed: Sep. 25, 2025. [Online]. Available: <https://github.com/google/bindiff>
- [33] Vector 35, Inc., "Signature Kit Plugin." Accessed: Sep. 17, 2025. [Online]. Available: <https://github.com/Vector35/sigkit>
- [34] K. M. Nakagawa, "binja-xpr-analyzer." Accessed: Sep. 17, 2025. [Online]. Available: <https://github.com/FFRI/binja-xpr-analyzer>
- [35] H. Oakley, "Watch your background: background activities with DAS-CTS." Accessed: Sep. 16, 2025. [Online]. Available: <https://eclecticlight.co/2025/01/29/watch-your-background-background-activities-with-das-cts/>
- [36] A. Schmidt, "XPR-dump: Helper scripts to automate the extraction of YARA rules from XProtectRemediators." Accessed: Sep. 10, 2025. [Online]. Available: <https://github.com/alid3ns/XPR-dump>
- [37] K. M. Nakagawa, "XProtect Remediator 'secret' configurations." Accessed: Sep. 10, 2025. [Online]. Available: <https://github.com/FFRI/XPRSecretConfigs>
- [38] C. Long, "osquery-xpdb." Accessed: Sep. 15, 2025. [Online]. Available: <https://clo.ng/blog/osquery-xpdb/>
- [39] H. Oakley, "What do XProtect BehaviourService and Bastion rules do?" Accessed: Sep. 15, 2025. [Online]. Available: <https://eclecticlight.co/2024/06/28/what-do-xprotect-behaviourservice-and-bastion-rules-do/>
- [40] R. Avni and O. Davidov, "DNS Hijacking: A New Method of MITM Attack Observed in the Wild." Accessed: Sep. 15, 2025. [Online]. Available: <https://web.archive.org/web/20200102160037/https://www.airov.com/dns-hijacking-a-new-method-of-mitm-attack-observed-in-the-wild/>
- [41] H. Oakley, "XProCheck: a new utility to inspect anti-malware scans." Accessed: Sep. 15, 2025. [Online]. Available: <https://eclecticlight.co/2022/09/05/xprocheck-a-new-utility-to-inspect-anti-malware-scans/>
- [42] J. McCall and D. Gregor, "SE-0289: Result Builders." Accessed: Sep. 16, 2025. [Online]. Available: <https://github.com/swiftlang/swift-evolution/blob/main/proposals/0289-result-builders.md>

- [43] C. Katri, “awesome-result-builders.” Accessed: Sep. 16, 2025. [Online]. Available: <https://github.com/carson-katri/awesome-result-builders>
- [44] K. M. Nakagawa, “RemediationBuilder DSL Specification.” Accessed: Sep. 16, 2025. [Online]. Available: <https://github.com/FFRI/RemediationBuilderDSLSpec>
- [45] K. M. Nakagawa, “XPR Test Suite.” Accessed: Sep. 16, 2025. [Online]. Available: <https://github.com/FFRI/XPRTTestSuite>
- [46] P. Stokes, “How AdLoad macOS Malware Continues to Adapt & Evade.” Accessed: Sep. 16, 2025. [Online]. Available: <https://www.sentinelone.com/labs/how-adload-macos-malware-continues-to-adapt-evade/>
- [47] P. Stokes, “Massive New Adload Campaign Goes Entirely Undetected by Apple’s XProtect.” Accessed: Sep. 16, 2025. [Online]. Available: <https://www.sentinelone.com/labs/massive-new-adload-campaign-goes-entirely-undetected-by-apples-xprotect/>
- [48] The MITRE Corporation, “MITRE ATT&CK: Bundlore.” Accessed: Sep. 25, 2025. [Online]. Available: <https://attack.mitre.org/software/S0482/>
- [49] L. Magisa and Q. Sun, “Pilfered Keys: Free App Infected by Malware Steals Keychain Data.” Accessed: Sep. 16, 2025. [Online]. Available: https://www.trendmicro.com/en_us/research/22/k/pilfered-keys-free-app-infected-by-malware-steals-keychain-data.html
- [50] T. Fakterman, “Through the Cortex XDR Lens: macOS Pirrit Adware.” Accessed: Sep. 23, 2025. [Online]. Available: <https://www.paloaltonetworks.com/blog/security-operations/through-the-cortex-xdr-lens-macos-pirrit-adware/>
- [51] T. Lambert, “Analyzing Pirrit Adware Installer.” Accessed: Sep. 23, 2025. [Online]. Available: <https://forensicitguy.github.io/analyzing-pirrit-adware-installer/>
- [52] L. B. Georgy Kucherin and I. Kuznetsov, “Dissecting TriangleDB, a Triangulation spyware implant.” Accessed: Sep. 16, 2025. [Online]. Available: <https://securelist.com/triangledb-triangulation-implant/110050/>
- [53] M. K. Anton V. Ivanov and I. Mogilin, “Shlayer Trojan attacks one in ten macOS users.” Accessed: Sep. 16, 2025. [Online]. Available: <https://securelist.com/shlayer-for-macos/95724/>
- [54] M.-E. M. Léveillé, “I see what you did there: A look at the CloudMensis macOS spyware.” Accessed: Sep. 16, 2025. [Online]. Available: <https://www.welivesecurity.com/2022/07/19/i-see-what-you-did-there-look-cloudmensis-macos-spyware/>
- [55] F. M. Sidera and O. Caspi, “Mac Systems Turned into Proxy Exit Nodes by Adload.” Accessed: Sep. 16, 2025. [Online]. Available: <https://levelblue.com/blogs/labs-research/mac-systems-turned-into-proxy-exit-nodes-by-adload>
- [56] H. Oakley, “Apple declares war on Adload malware.” Accessed: Sep. 17, 2025. [Online]. Available: <https://eclecticlight.co/2024/04/25/apple-declares-war-on-adload-malware/>
- [57] P. Stokes, “How AdLoad macOS Malware Continues to Adapt & Evade.” Accessed: Sep. 17, 2025. [Online]. Available: <https://www.sentinelone.com/labs/how-adload-macos-malware-continues-to-adapt-evade/>

- [58] C. Fitzl, "Beyond the good ol' LaunchAgents - 4 - cron jobs." Accessed: Sep. 23, 2025. [Online]. Available: https://theevilbit.github.io/beyond/beyond_0004/
- [59] S. Mayers and C. Lopez, "How AMOS macOS Stealer Avoids Detection." Accessed: Sep. 17, 2025. [Online]. Available: <https://www.kandji.io/blog/amos-macos-stealer-analysis>
- [60] J. Levin, *MacOS and iOS Internals, Volume I: User Mode*. United States: Technologic Press, 2017. Available: <https://www.amazon.com/dp/099105556X>
- [61] Apple Developer Forums, Thread 742828, "XProtect Remediator and BadGacha." Accessed: Sep. 17, 2025. [Online]. Available: <https://developer.apple.com/forums/thread/742828>
- [62] M. Datka, "CrowdStrike Uncovers New MacOS Browser Hijacking Campaign." Accessed: Sep. 17, 2025. [Online]. Available: <https://www.crowdstrike.com/en-us/blog/how-crowdstrike-uncovered-a-new-macos-browser-hijacking-campaign/>
- [63] A. Waichulis, "Security Bite: Self-destructing macOS malware strain disguised as legitimate Mac app." Accessed: Sep. 17, 2025. [Online]. Available: <https://9to5mac.com/2024/02/29/security-bite-self-destructing-macos-malware-strain-disguised-as-legitimate-mac-app/>
- [64] A. Kohler and C. Lopez, "Malware: Cuckoo Behaves Like Cross Between Infostealer and Spyware." Accessed: Sep. 17, 2025. [Online]. Available: <https://www.kandji.io/blog/malware-cuckoo-infostealer-spyware>
- [65] Darktrace Holdings Ltd., "Darktrace Detection of 3CX Supply Chain Attack." Accessed: Sep. 16, 2025. [Online]. Available: <https://www.darktrace.com/blog/3cx-supply-chain-compromise-how-darktrace-uncovered-a-smooth-operator>
- [66] Apple, Inc., "About Apple threat notifications and protecting against mercenary spyware." Accessed: Sep. 23, 2025. [Online]. Available: <https://support.apple.com/en-us/102174>
- [67] P. Wardle, "Mirror Mirror: Restoring Reflective Code Loading on macOS," in *Objective by the Sea (OBTS) v7.0*, Dec. 2024. Accessed: Sep. 10, 2025. [Online]. Available: https://objectivebythesea.org/v7/talks/OBTS_v7_pWardle.pdf
- [68] P. Wardle, *The Art of Mac Malware, Volume 2: Detecting Malicious Software*. United States: No Starch Press, 2025. Available: <https://www.amazon.com/dp/1718503784/>
- [69] FFRI Security, Inc., "PoC-public." Accessed: Sep. 17, 2025. [Online]. Available: <https://github.com/FFRI/PoC-public>
- [70] O. Yair, "Aikido: Turning EDRs to Malicious Wipers Using 0-day Exploits," in *Black Hat Europe 2022 Briefings*, Informa PLC, Dec. 2022. Accessed: Sep. 21, 2025. [Online]. Available: <https://www.blackhat.com/eu-22/briefings/schedule/#aikido-turning-edrs-to-malicious-wipers-using--day-exploits-29336>
- [71] P. Wardle, "Demystifying (& Bypassing) macOS's Background Task Management," in *DEF CON 31*, Aug. 2023. Accessed: Sep. 25, 2025. [Online]. Available: <https://speakerdeck.com/patrickwardle/demystifying-and-bypassing-macoss-background-task-management>
- [72] C. Suter, "WrMeta." Accessed: Sep. 15, 2025. [Online]. Available: <https://groups.google.com/g/darwin-dev/c/7F6uth1rhKw/m/SJQ3zWxelgEJ>

- [73] “FreeBSD/Linux Kernel Cross Reference — hfs_cnode.c.” http://fxr.watson.org/fxr/source/bsd/hfs/hfs_cnode.c?v=xnu-1456.1.26;im=10#L967.
- [74] H. Oakley, “Ventura has changed app quarantine with a new xattr.” Accessed: Sep. 17, 2025. [Online]. Available: <https://eclecticlight.co/2023/03/13/ventura-has-changed-app-quarantine-with-a-new-xattr/>
- [75] K. M. Nakagawa, “Provenance Checker.” Accessed: Sep. 10, 2025. [Online]. Available: <https://github.com/FFRI/ProvenanceChecker>
- [76] K. M. Nakagawa, “ShowProvenanceInfo.” Accessed: Sep. 25, 2025. [Online]. Available: <https://github.com/FFRI/ShowProvenanceInfo>
- [77] K. M. Nakagawa, “Bypassing macOS Security and Privacy Mechanisms: From Gatekeeper to System Integrity Protection,” in *CODE BLUE 2023*, Nov. 2023. Accessed: Sep. 21, 2025. [Online]. Available: https://archive.codeblue.jp/2023/en/result/?content=Koh_Nakagawa
- [78] K. M. Nakagawa, “OpenRemediationBuilder.” Accessed: Sep. 17, 2025. [Online]. Available: <https://github.com/FFRI/OpenRemediationBuilder>