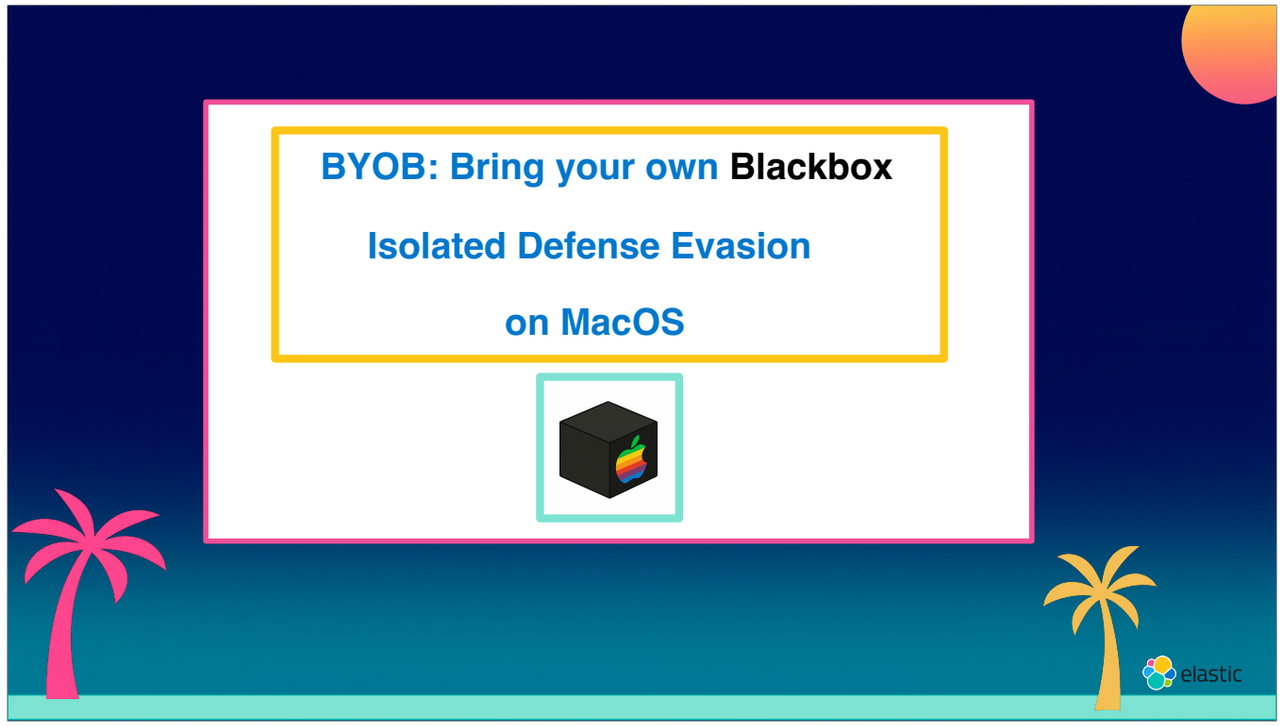
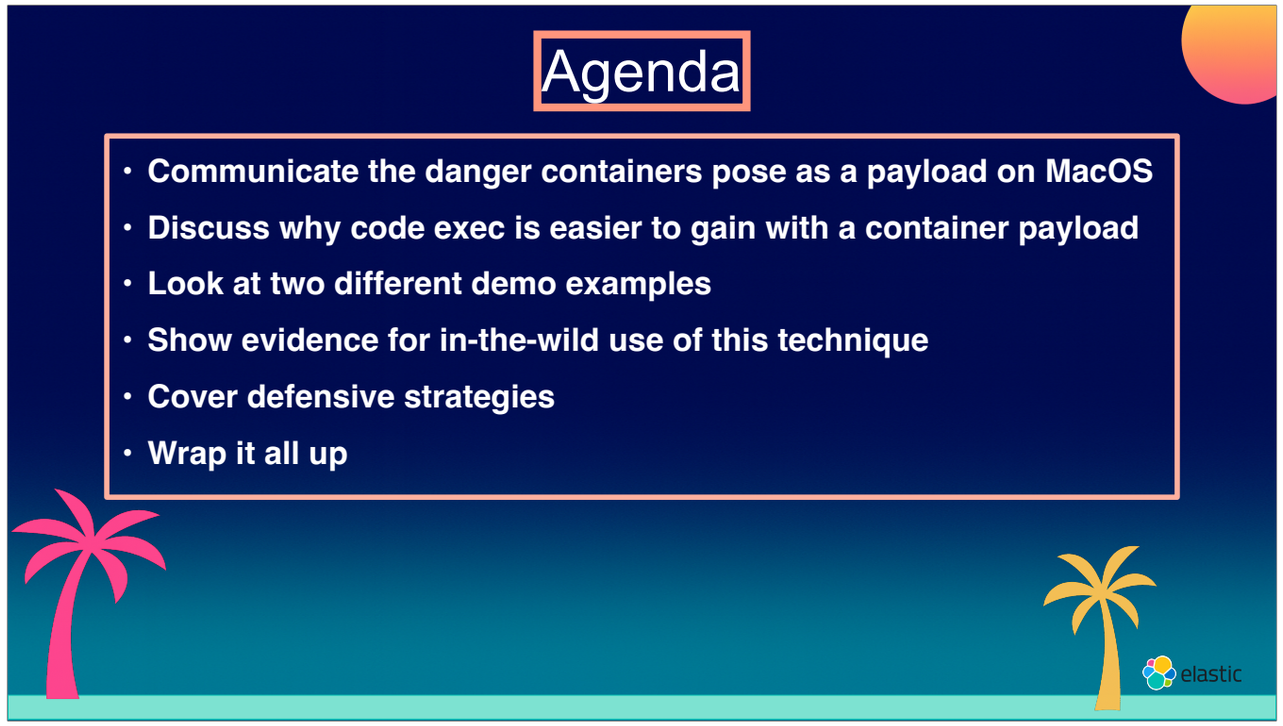


This is BYOB: Bring your own Blackbox Isolated Defense Evasion on MacOS.

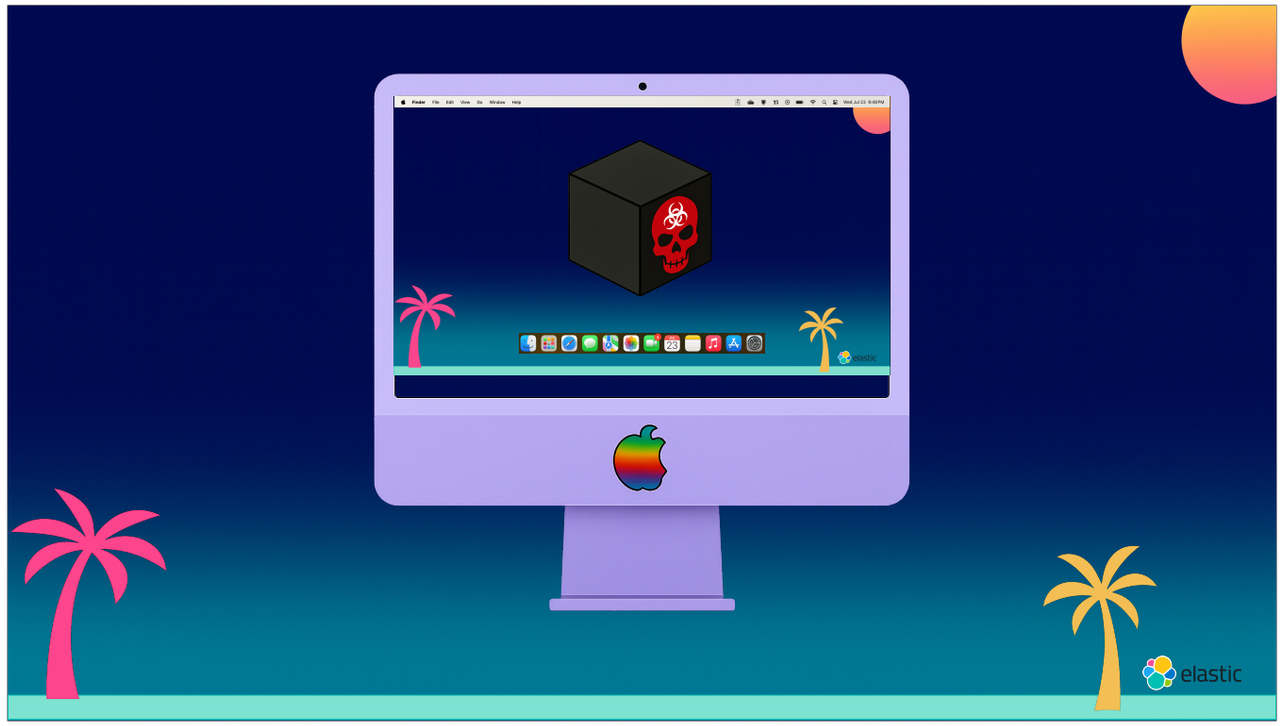


My name is Colson Wilhoit, I’m currently a Senior Security Research Engineer at Elastic and an Army National Guard Cyber Protection Team officer for over a decade now. On X I’m @DefSecSentinel—you may know my work on threats like KandyKorn, JokerSpy, and RustBucket.

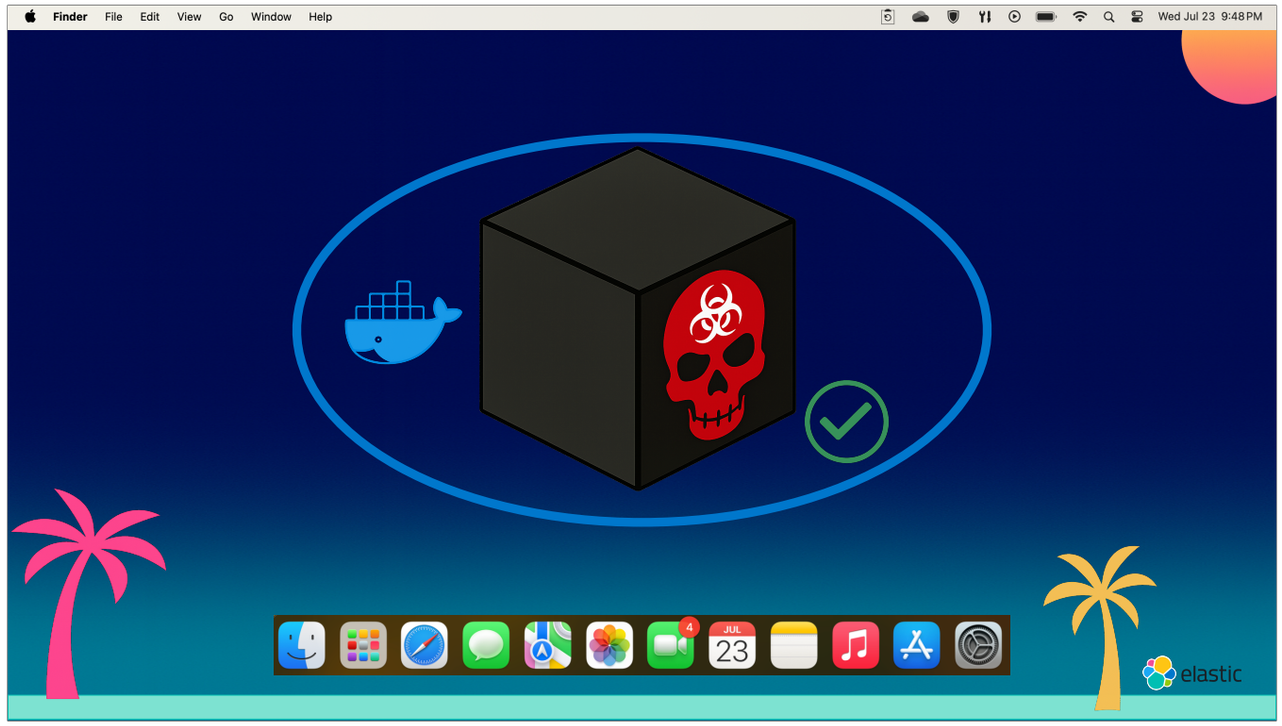


Here is everything we are going to cover today. Its pretty packed.

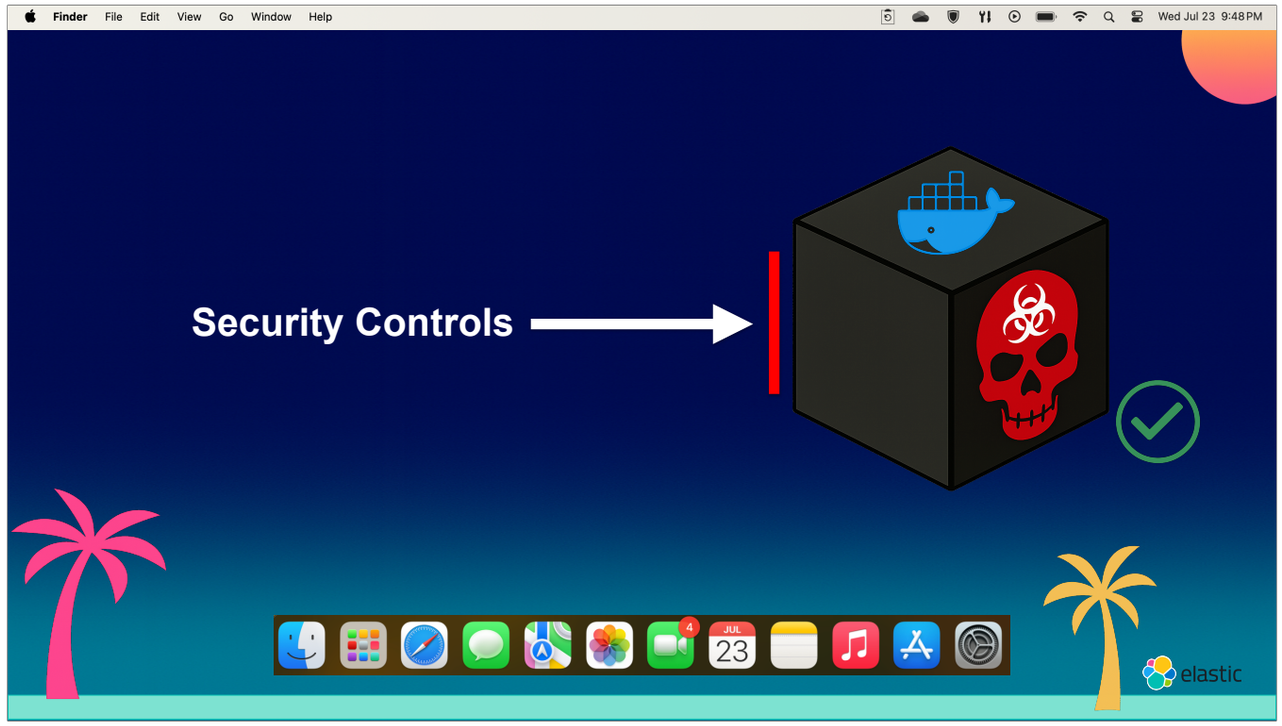
So lets get started



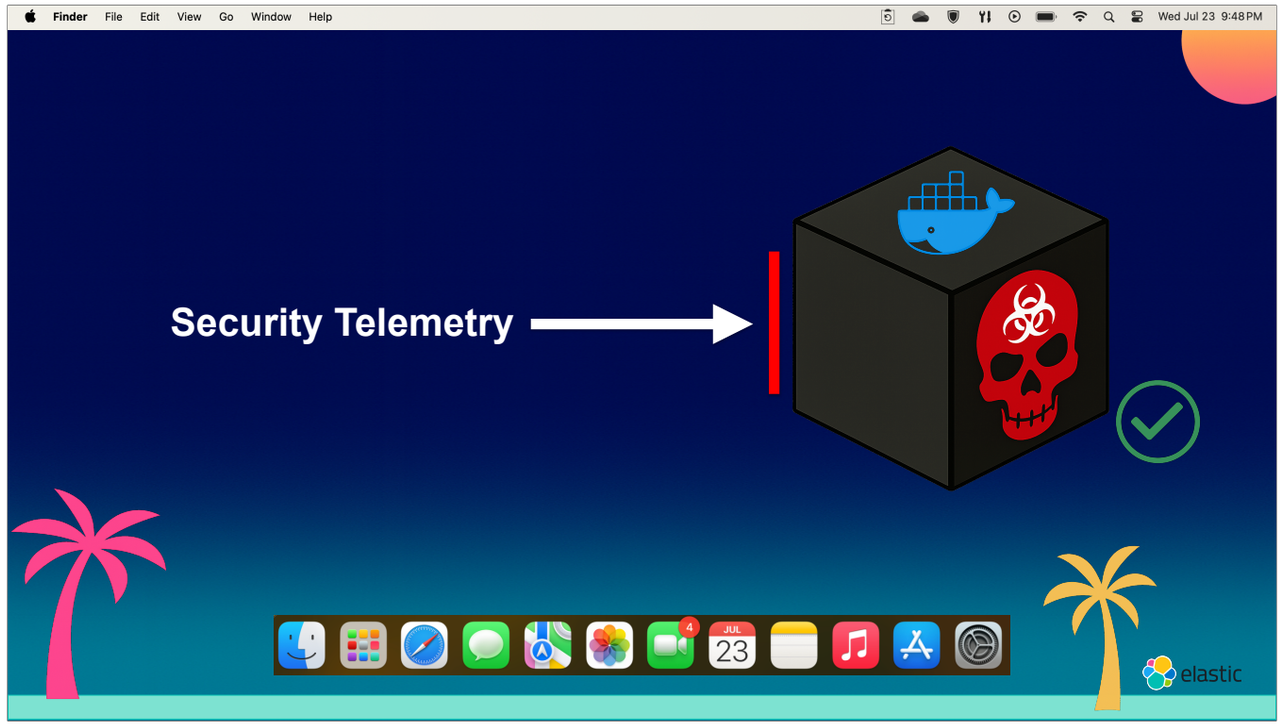
Alright, I’d like you to imagine you’re a threat actor who has gained code execution on a macOS target. What if you could run your second stage payload or post exploitation job



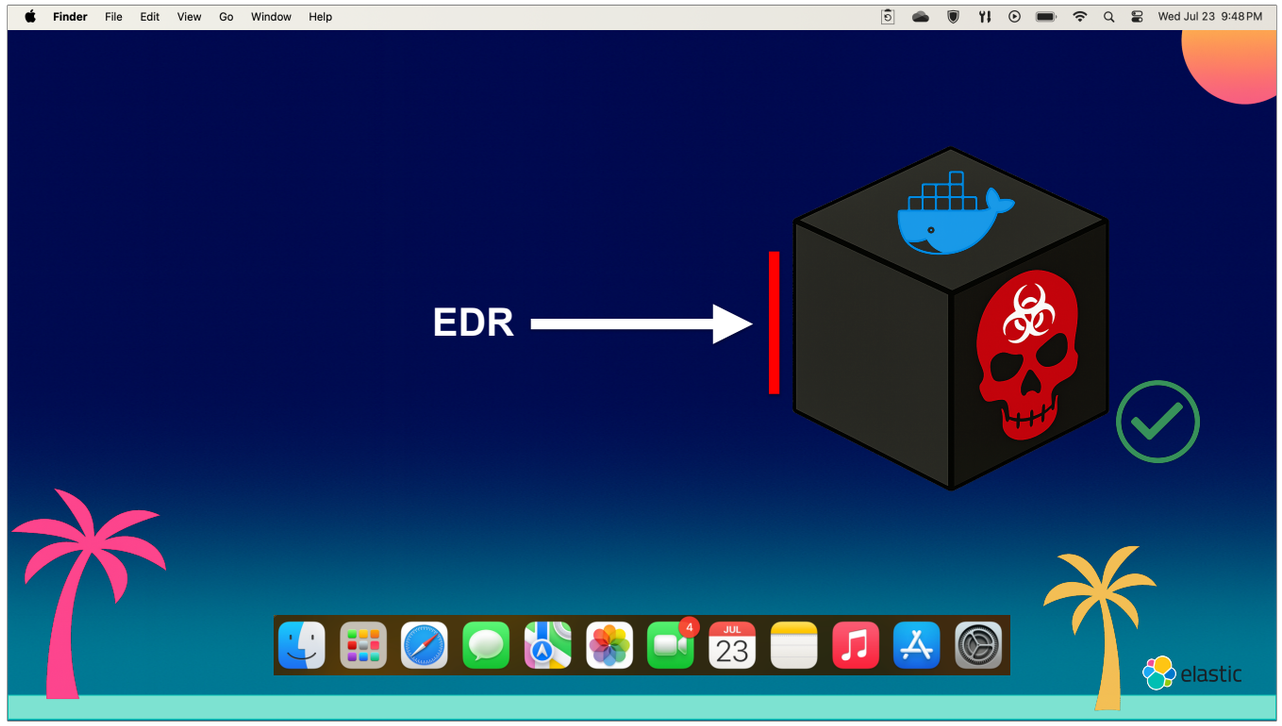
inside a well-known, signed, third-party process that macOS already trusts, one with generally all the privileges you need—that the vast majority of high-value targets use daily, and it provides anonymity against



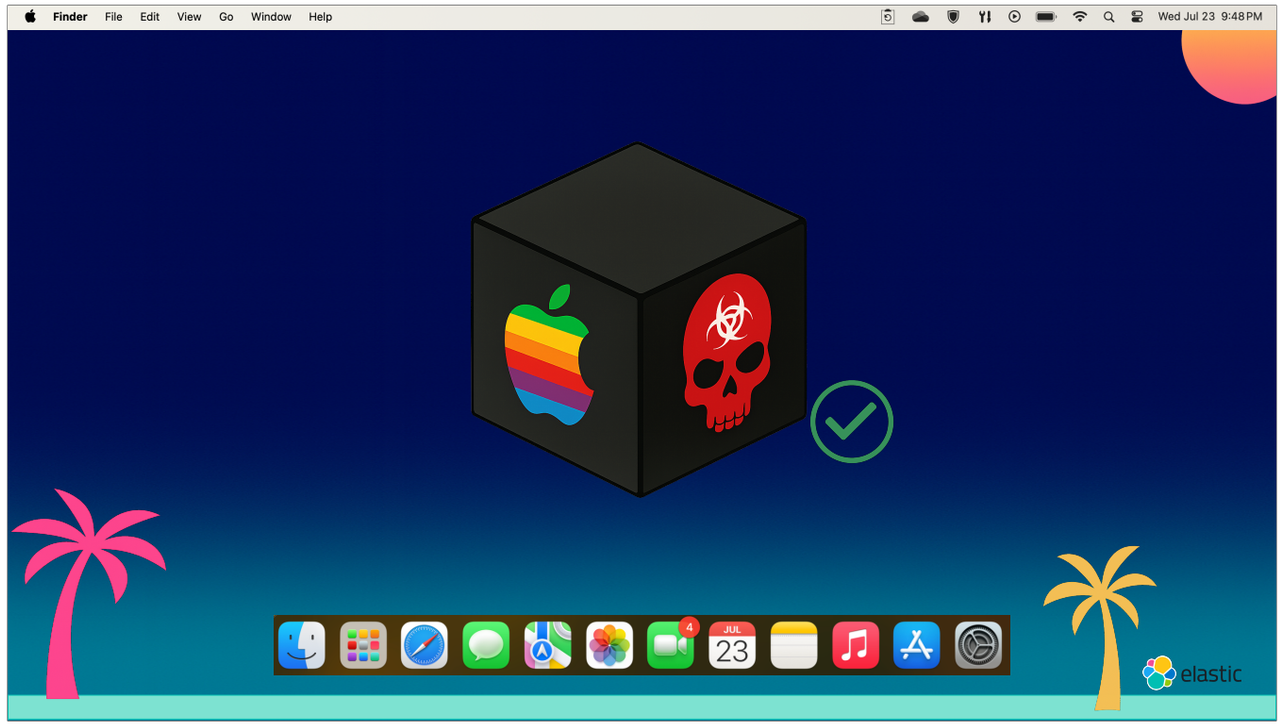
security controls like Gatekeeper, XProtect,



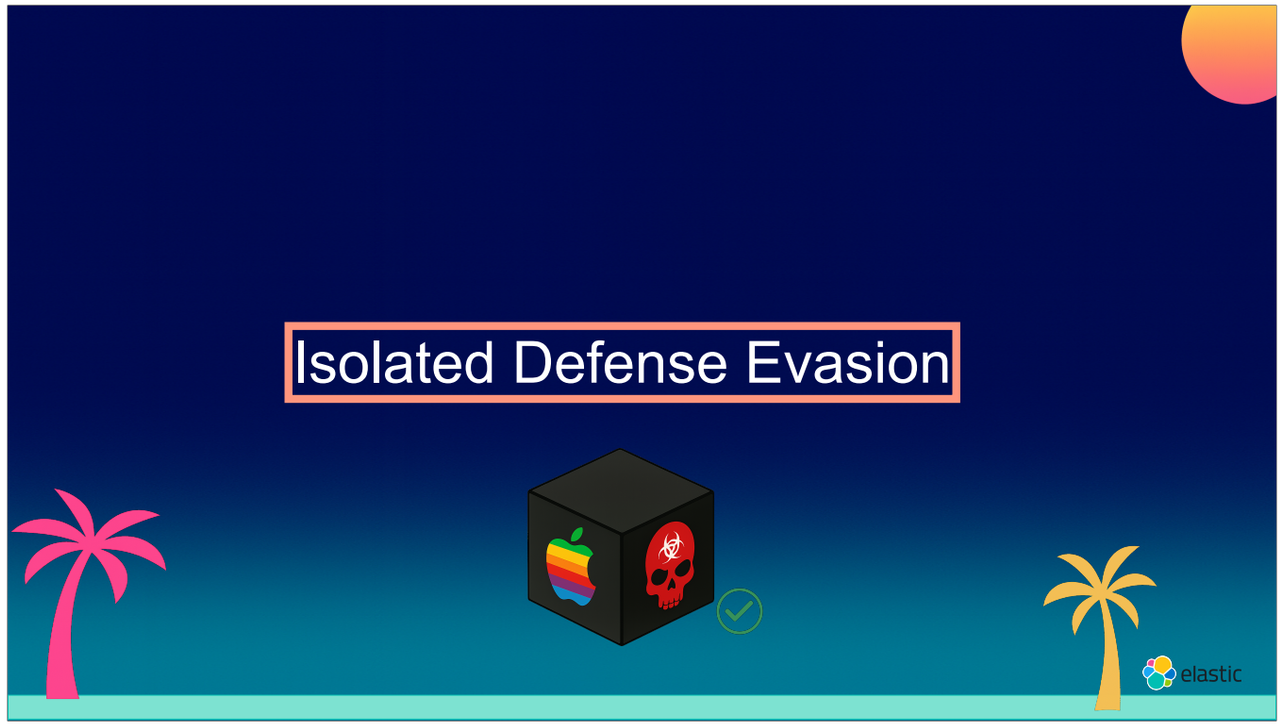
telemetry from the Endpoint Security Framework, and



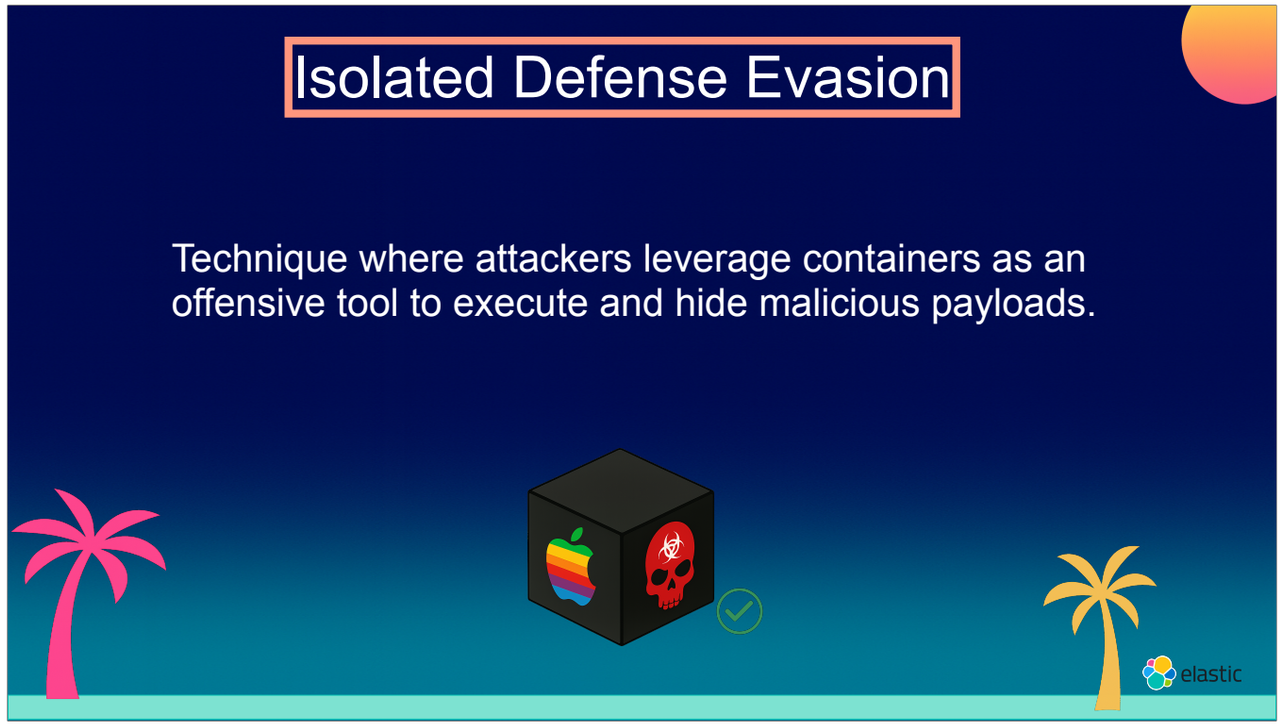
advanced endpoint detection and response solutions.



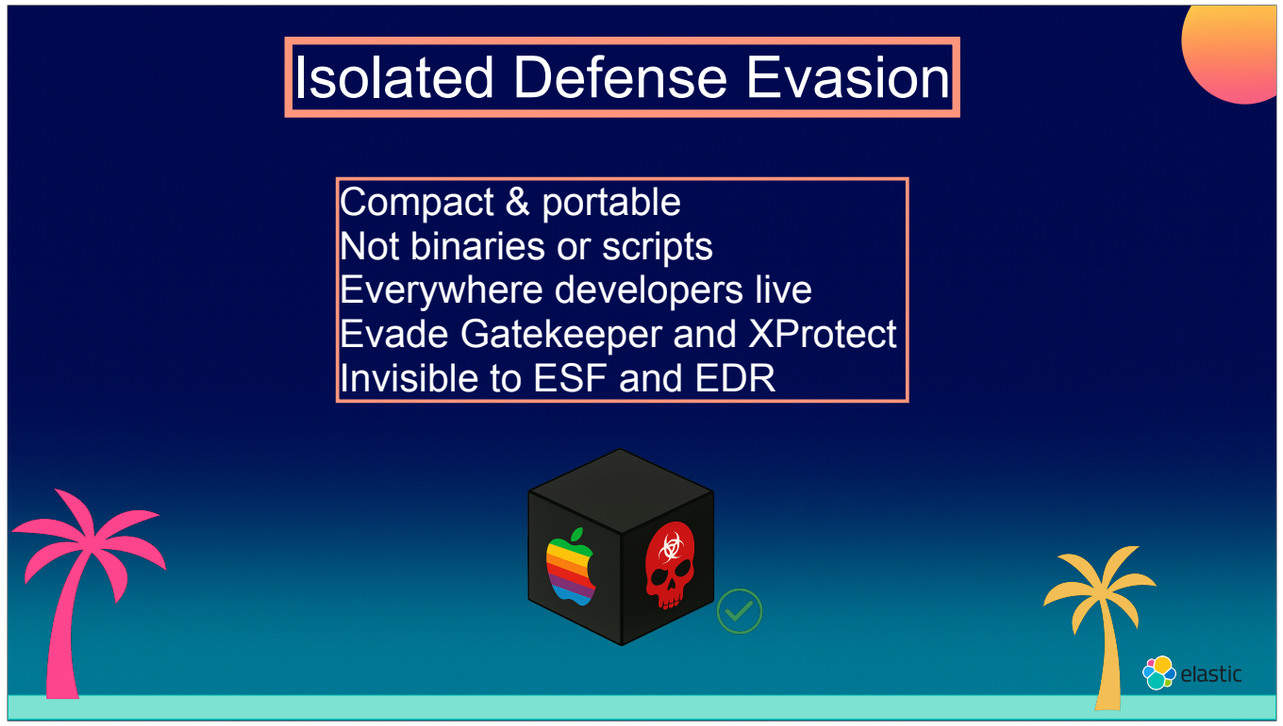
Now imagine Apple handing you a native, way to reap all those benefits, complete with well-documented APIs and within a signed, trusted system process. Well, no imagination required: it’s here in macOS Tahoe, with the new Container Framework.



What I'm talking about here is a container, and this technique is something I’m referring to as Isolated Defense Evasion. While most think of container security as preventing breakouts to the host, very few consider: what if the container itself is the payload?



What exactly is isolated defense evasion? At its core, it's a post-exploitation technique where attackers leverage containers and their frameworks not as a defensive sandbox or self-contained application, but as an offensive tool to execute and hide their malicious payloads.



What are some of the reasons this technique is so effective and powerful?

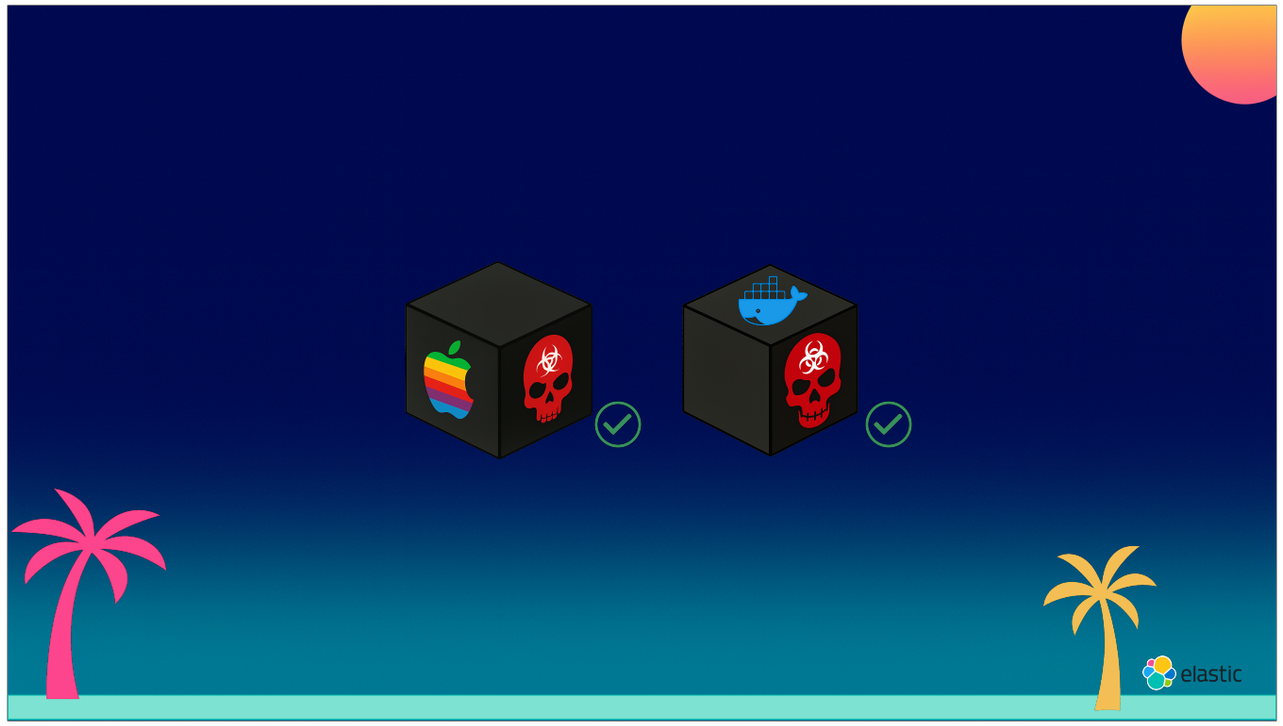
Containers are small and portable — you can smuggle a payload in a container image under 150 MB. Significantly smaller than a script or binary payload would be.

Container images are not binaries or scripts. They are essentially a filesystem snapshot stored as a compressed archive making them not subject to the checks or scans that binaries and scripts would be.

Containers live everywhere developers do — Docker is a trusted part of daily workflows at many companies across many industries today.

Activity inside a container completely evade Gatekeeper and XProtect because those checks don’t apply inside Linux VMs where these containers run and.

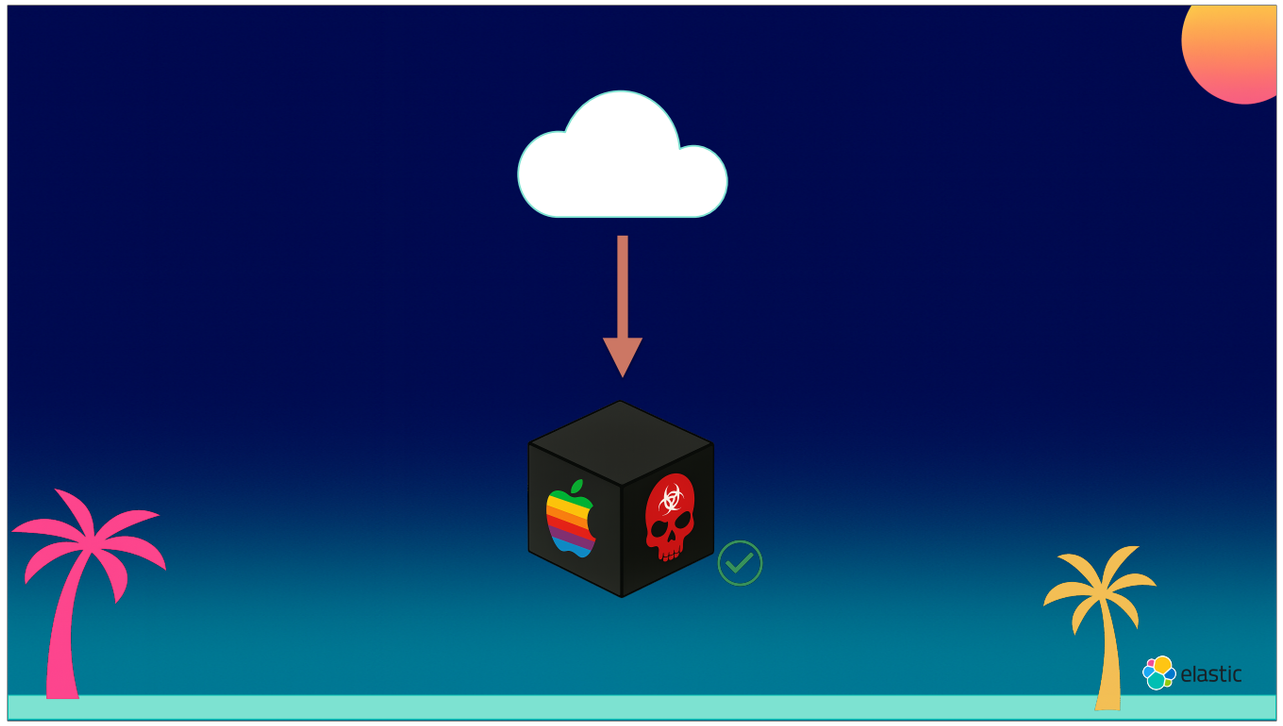
Containerized processes don’t traverse Apple’s Endpoint Security Framework, unless they touch the host. To EDR, what’s going on inside is invisible.



Let me address two likely questions you might have upfront.

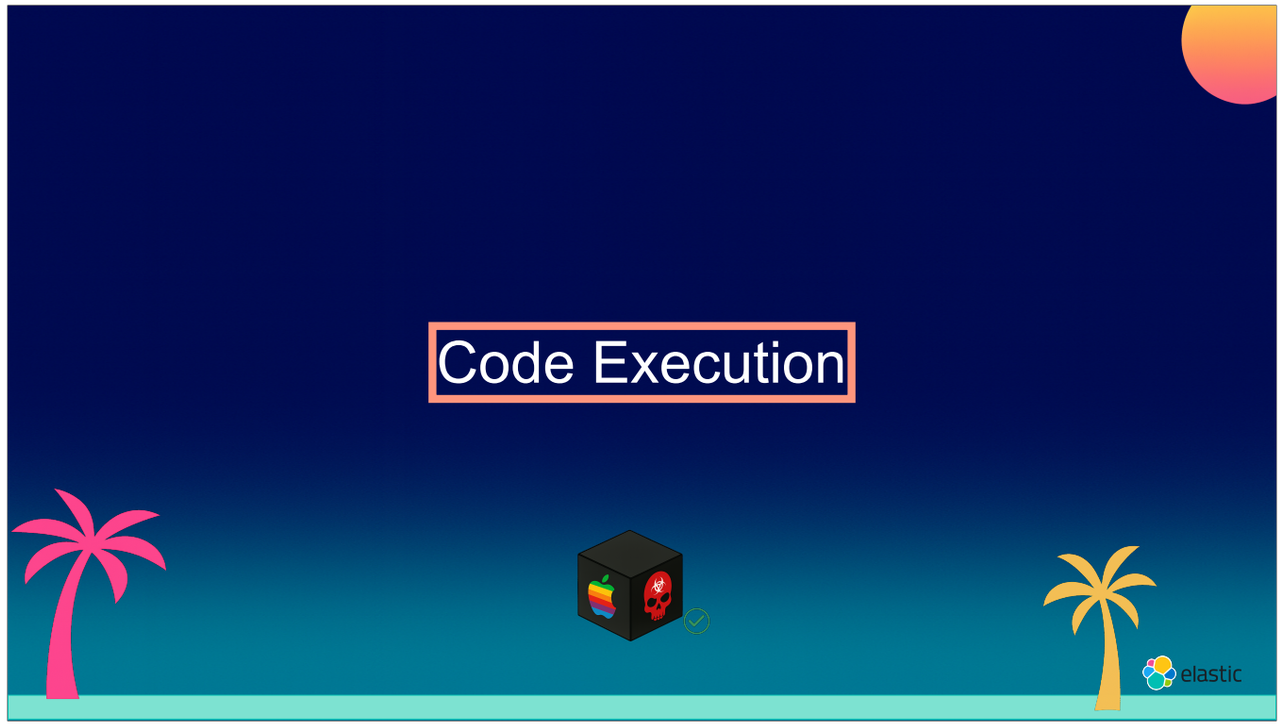
I mentioned using this technique relies on having Docker or in the case of Tahoe the Container Daemon present first and the bad guy can’t know those are present before hand right?

That’s true but in todays world with targeted attacks its possible to have a pretty high degree of certainty your victim is using one of these tools and at the very least you can include this technique as a solid stealthy option for once code execution is gained. This technique can be leveraged to sidestep defenses, gain privileges and achieve objectives without triggering alerts

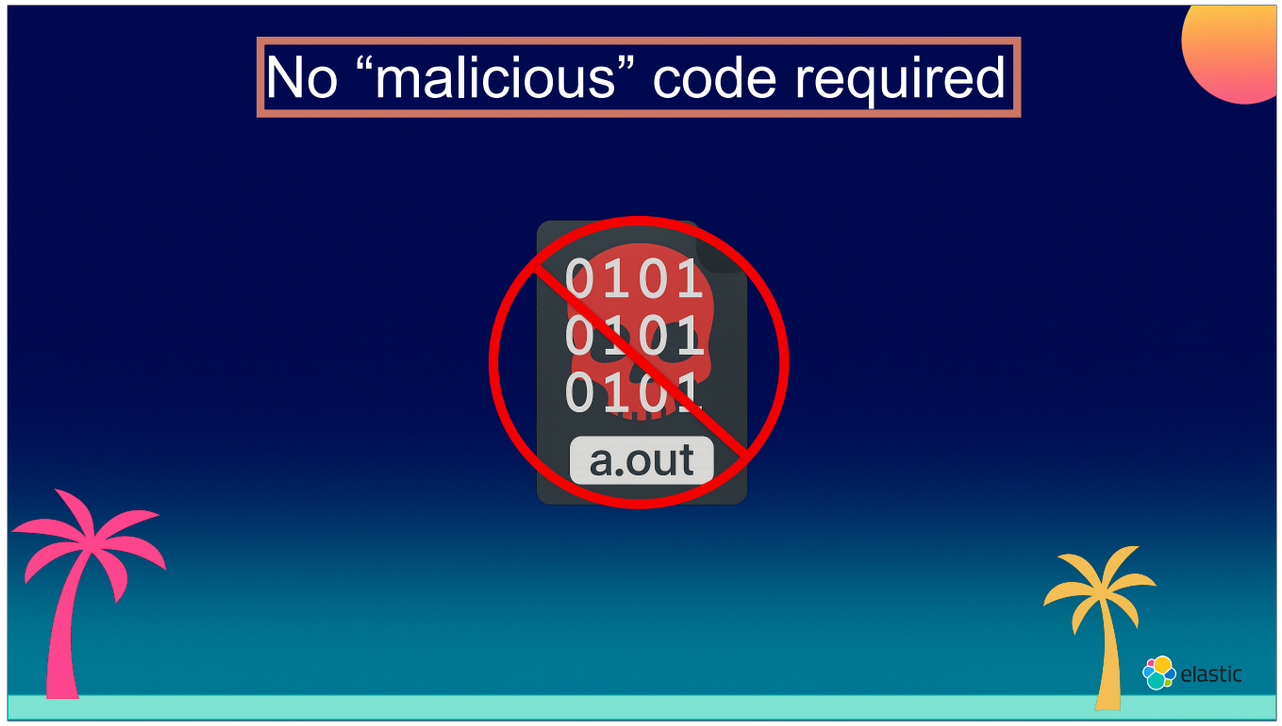


Now I also talked about using a Container as a payload which means you need some way to get the container onto the system in first place making initial access and some dropper or loader code execution a pre-requisite, right?

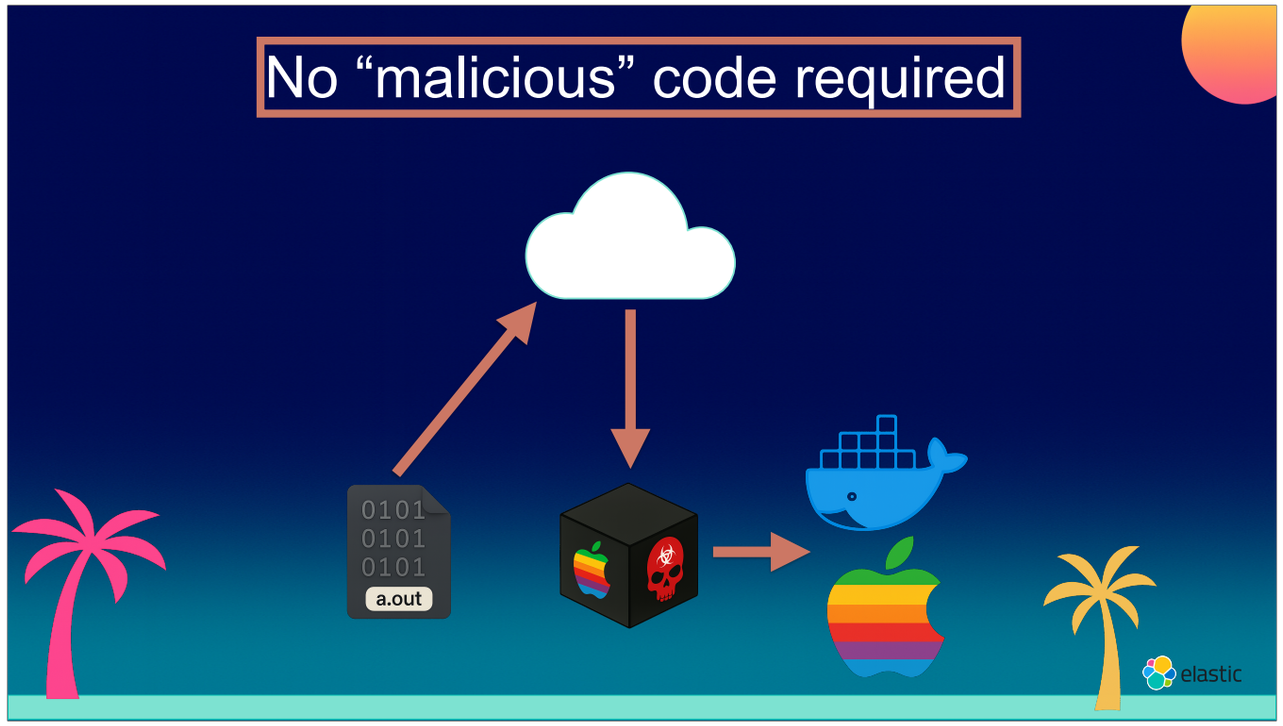
Lets talk about this in a little more detail.



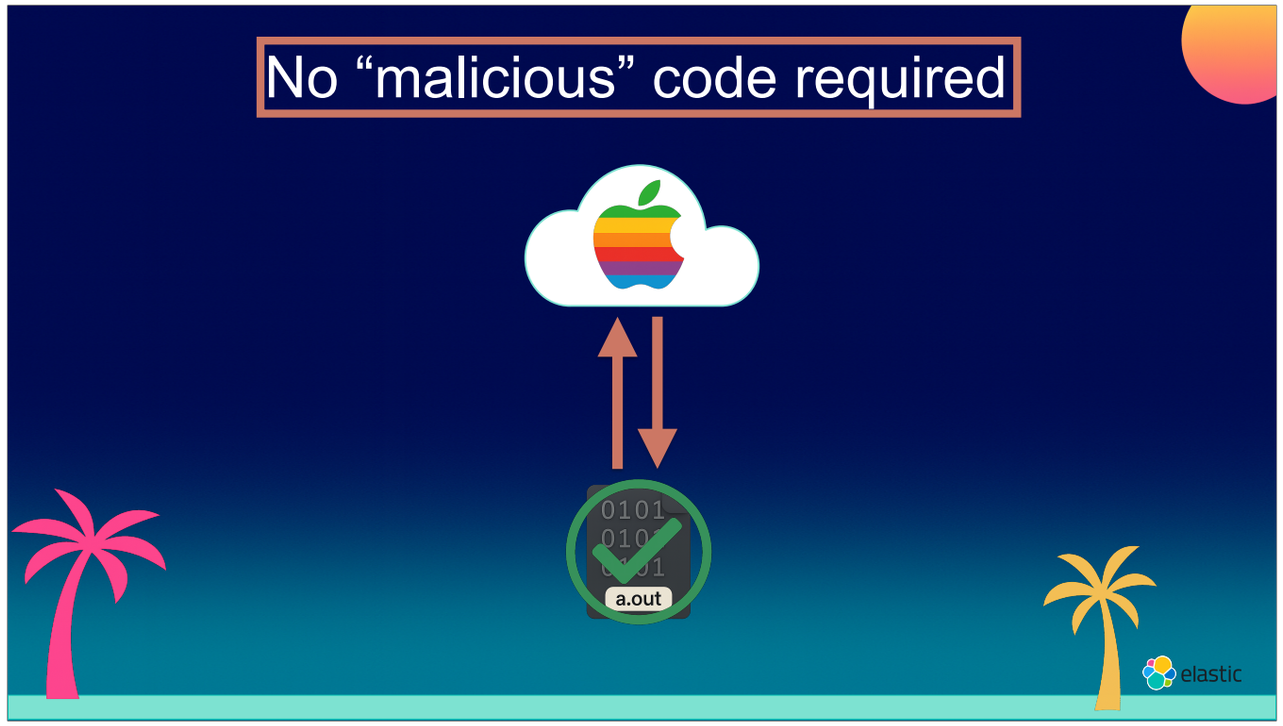
I think its important to pause real quick and touch on gaining code execution as it relates to using containers as payloads because, I believe, containers provide a unique opportunity that in my eyes elevates the danger of this technique.



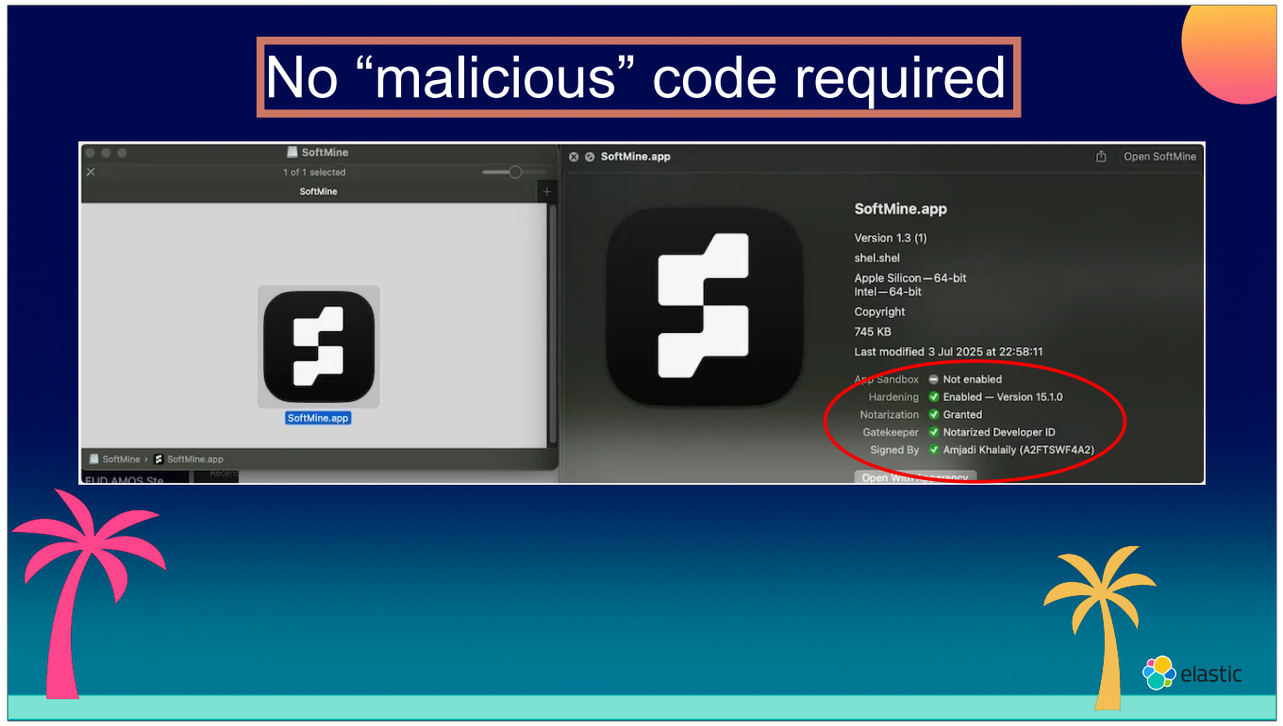
First off our dropper or loader payload, responsible for downloading the container image and orchestrating its execution, requires no code execution that would be considered inherently “malicious”.



Because all our payload needs to do is download the container image and use valid APIs to interact with Docker or the Container framework.



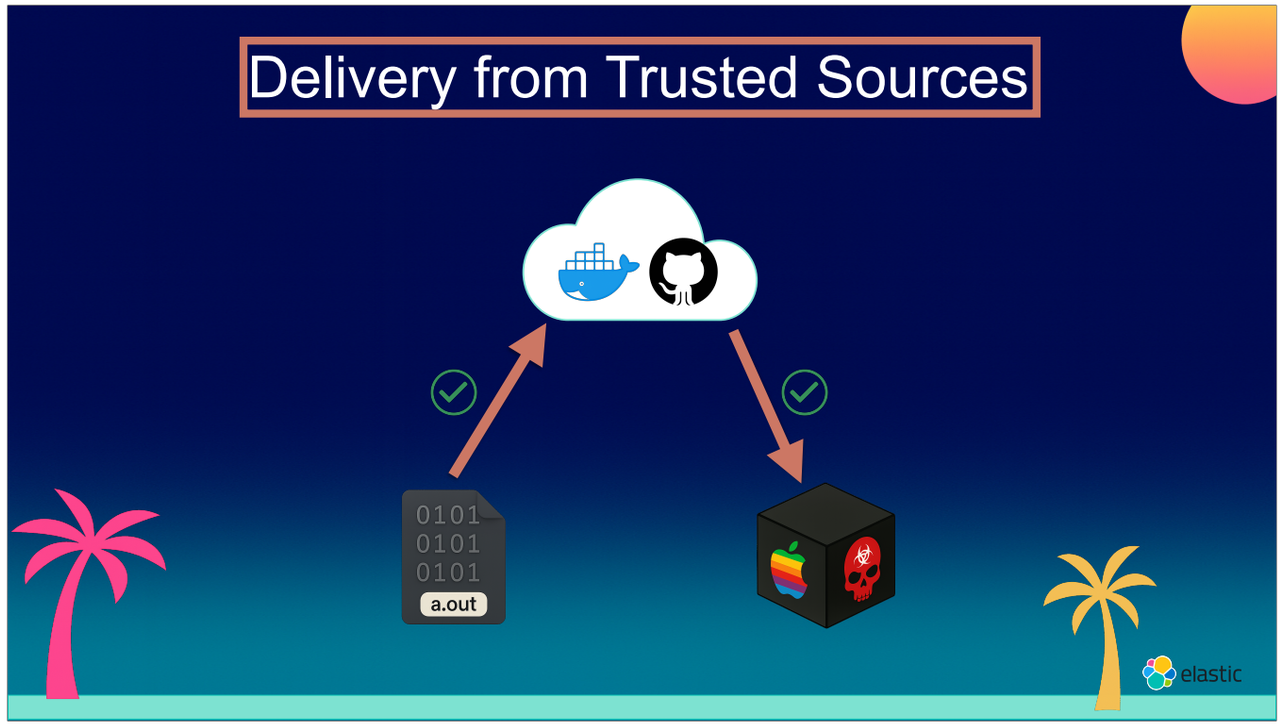
This provides us the ability to more easily and readily submit our payloads over to Apple to sign and notarize which will enable us to execute without being hindered by Gatekeeper or XProtect.



A really good example of this are some of the new information stealer samples, like Odyssey, that have recently been uncovered that were signed and notarized.

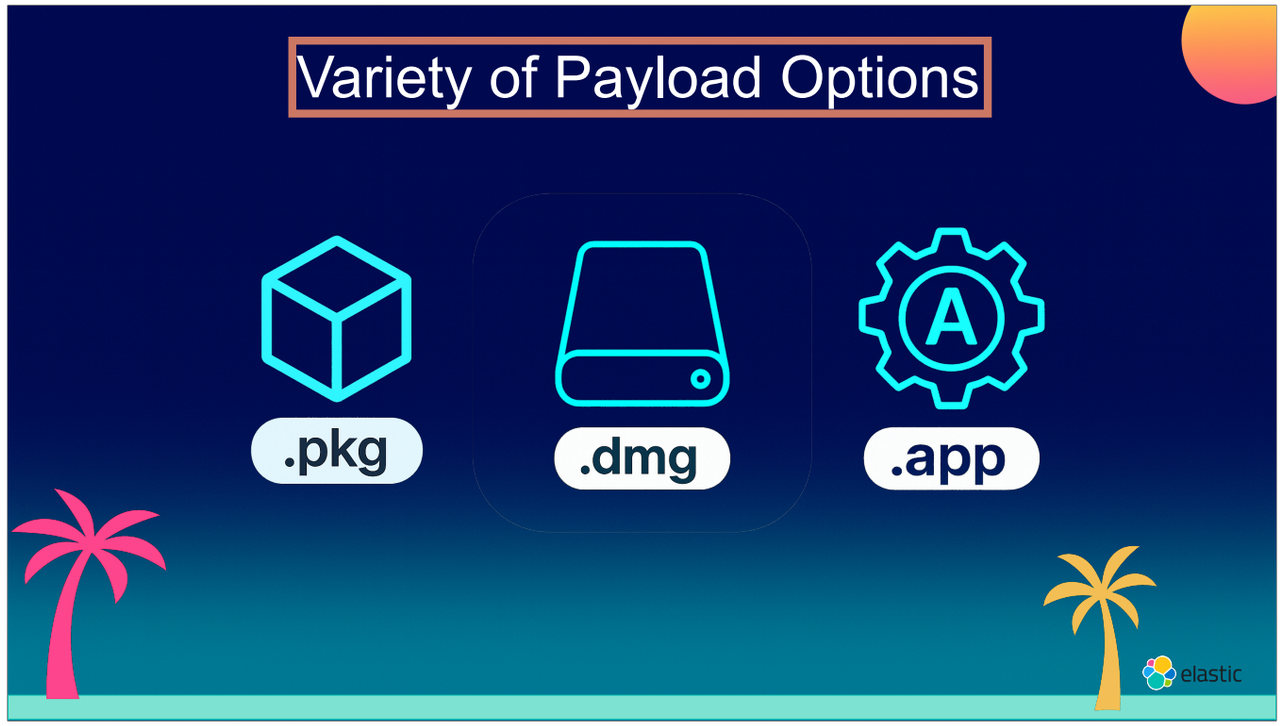
They used Swift to download and execute their Osascript payload as opposed to brining the script along with them.

Apple has really no reason not to sign or notarize because the app doesn’t do anything other than use the internet and execute a system binary. Apple has since revoked these certificates by the way.



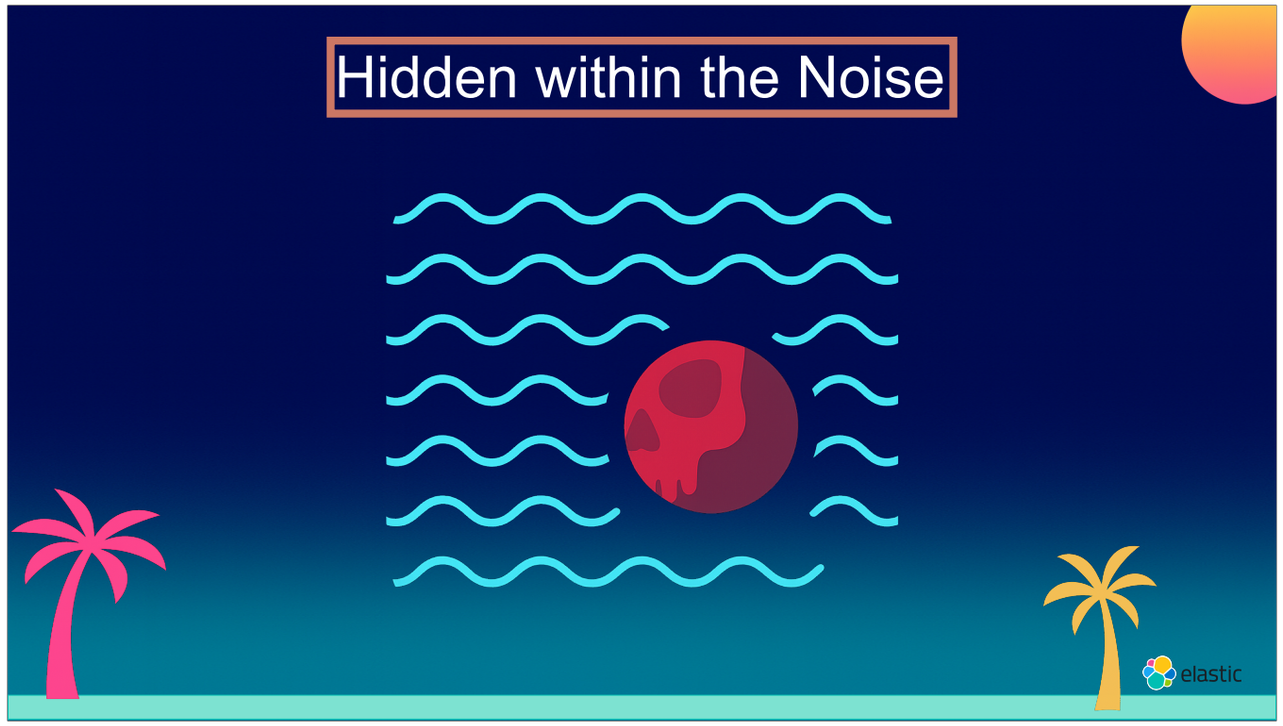
Second benefit that makes gaining code execution much easier in this case is the delivery via trusted sources.

We can host our container image on places like Docker Hub or Github which are trusted and used heavily by developers, engineers everyday ,almost guaranteeing a safe delivery.



Third benefit is the variety of payload delivery and execution options available. We could use almost any file format to deliver our payload: package, disk image, application to even include script packages like PyPi or NPM.

I could even deploy a PyPi package that downloads my container, runs it and cleanups



Fourth benefit and arguably the most important, in my opinion, is the ability for all of our activity with this technique to hide in the noise.

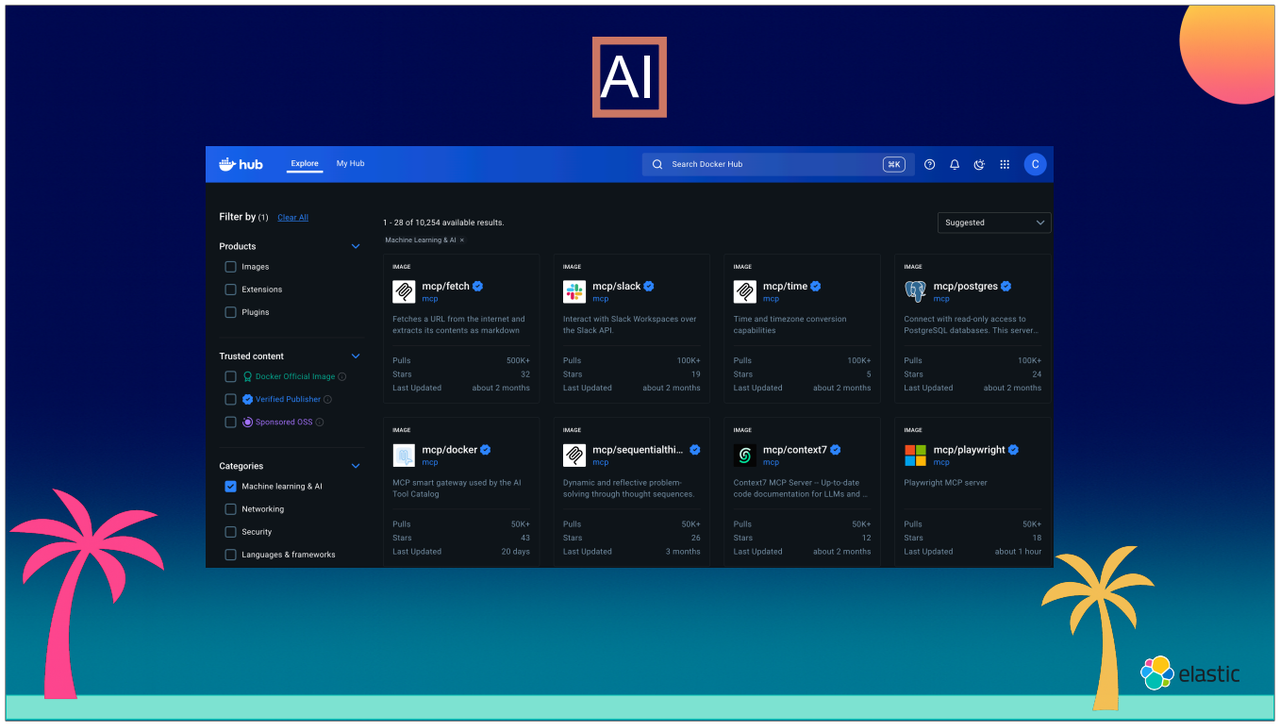
Developers and engineers do the exact same things our payload will do:

downloading of container images,
running container images,
accessing sensitive files from containers and more

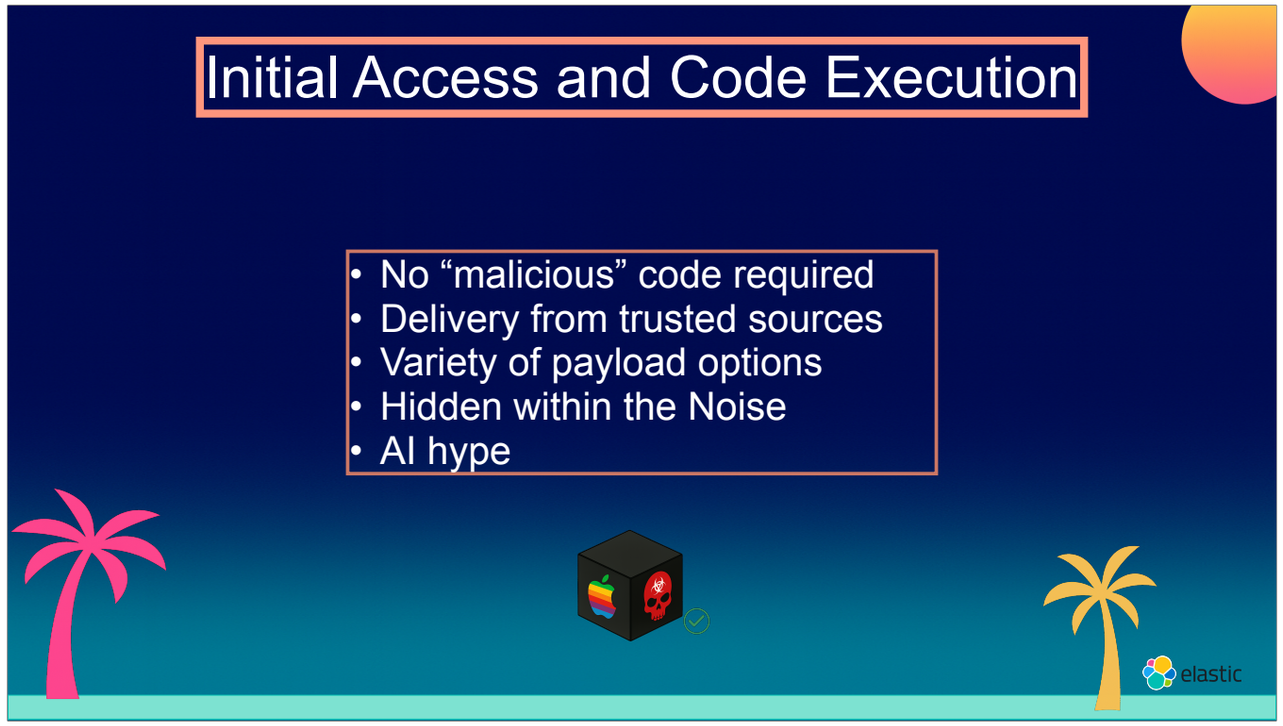
every single day, all the time.

Working for a big search and security company I can tell you this activity is very noisy and normal.

Our payload and all of its actions will fly under the radar blending in with normal activity



And last but not least I feel the need to mention “AI”. Containers are used heavily in the AI/ML space right now so using that as context provides a very nice way to slip in un-detected or without raising alarms. MCP servers or clients and Model containers provide a juicy opportunity for typo squatting, spoofing, etc..



Container images as a payload provide us with a higher degree of certainty, compared to most cases, in being able to gain initial access and code execution on a victims machine. Not fool proof but a payload that downloads a container image, loads and executes provides a unique opportunity.



Now let's see this in action

Enough Talk







elastic

Enough Talk





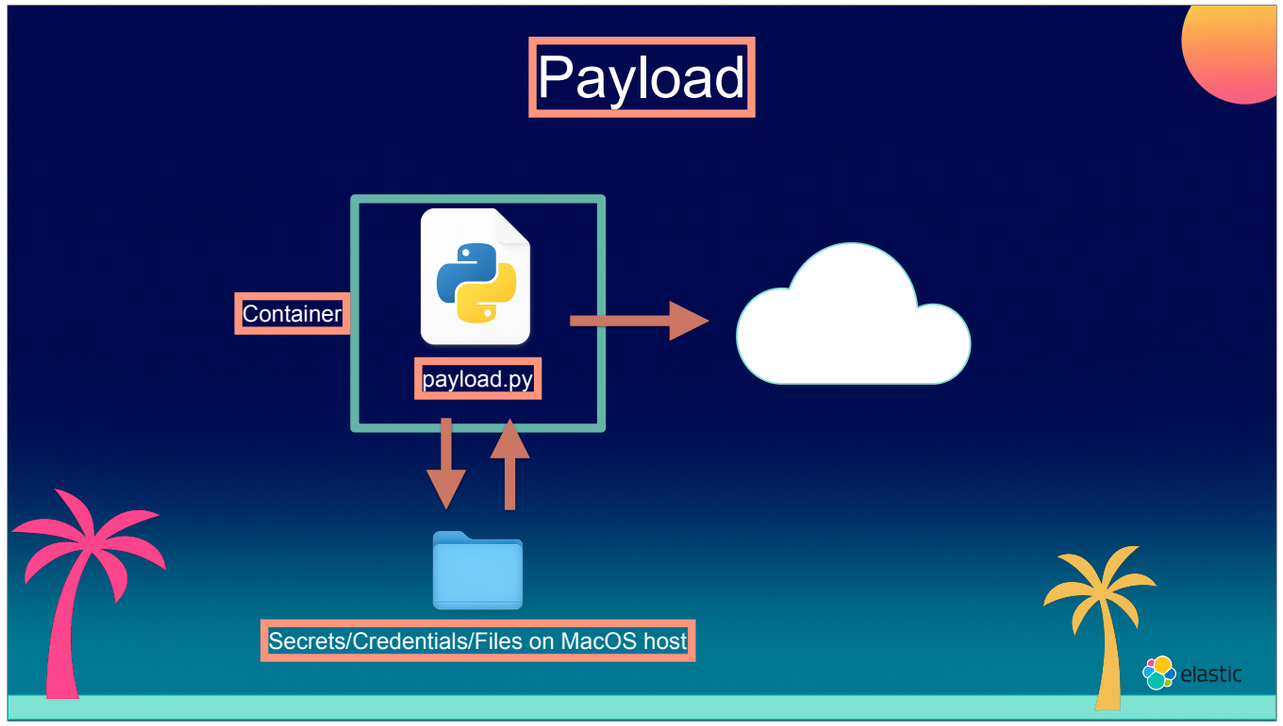


elastic



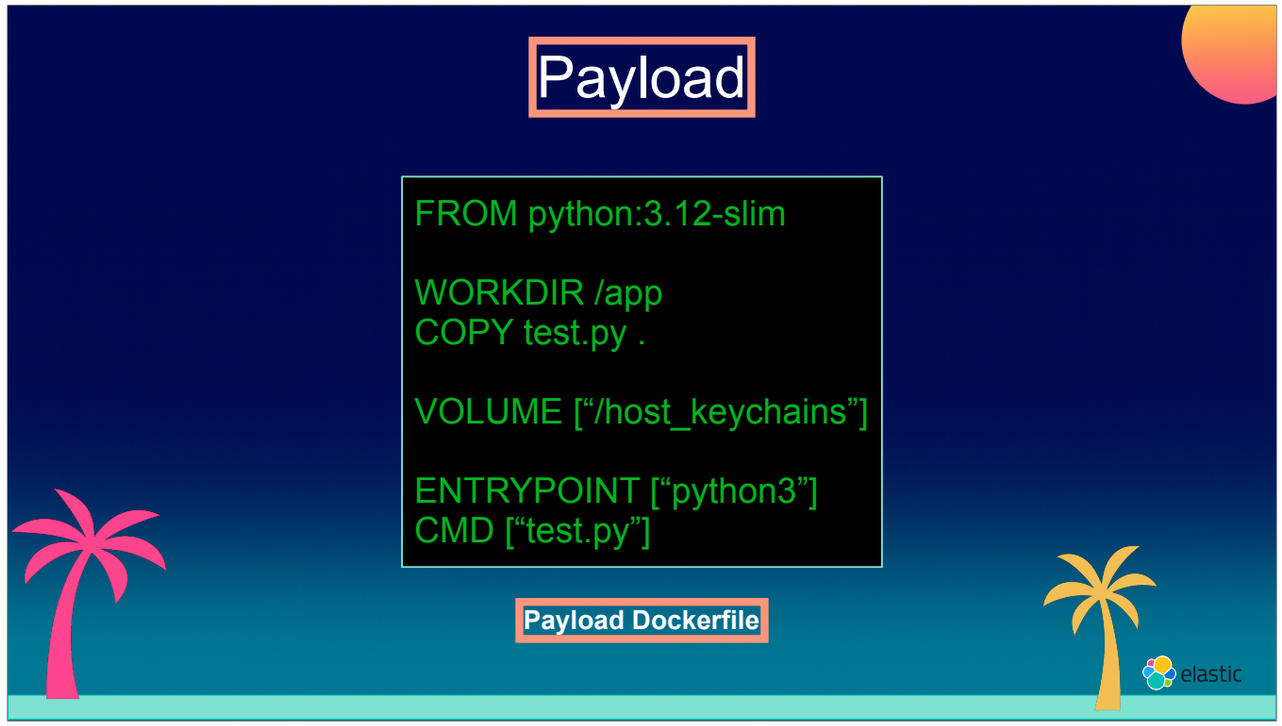
We are going to look at some concrete proof of concepts to clarify and prove what I've told you, showing what's possible. I've created two proof of concepts I will use to demonstrate an attack.

The first will target a macOS system pre-Tahoe on the latest version of Sequoia using Docker and the second will target a macOS system on the latest version of Tahoe utilizing the Container Framework.

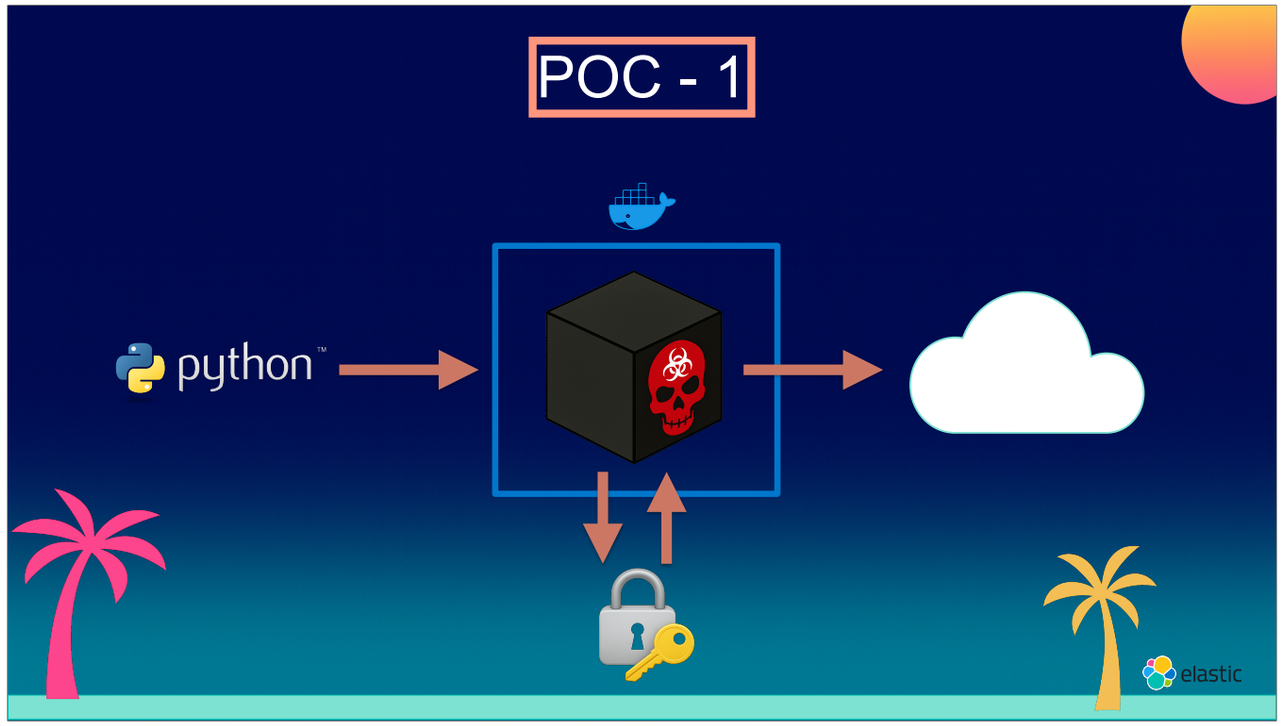


The containerized payload I’ve used in these POCs is a very small and very simple Python script that creates a compressed archive in memory of the mounted host folders contents, in this case the users keychain, and sends the archive to a remote file hosting provider.

The container simply acts as the payloads protection and executor.



And I’ll say the word payload is extremely generous here since the container is very basic, non-privileged and the Python script itself does things which are pretty standard for applications or app diagnostic collection not really malicious in nature.



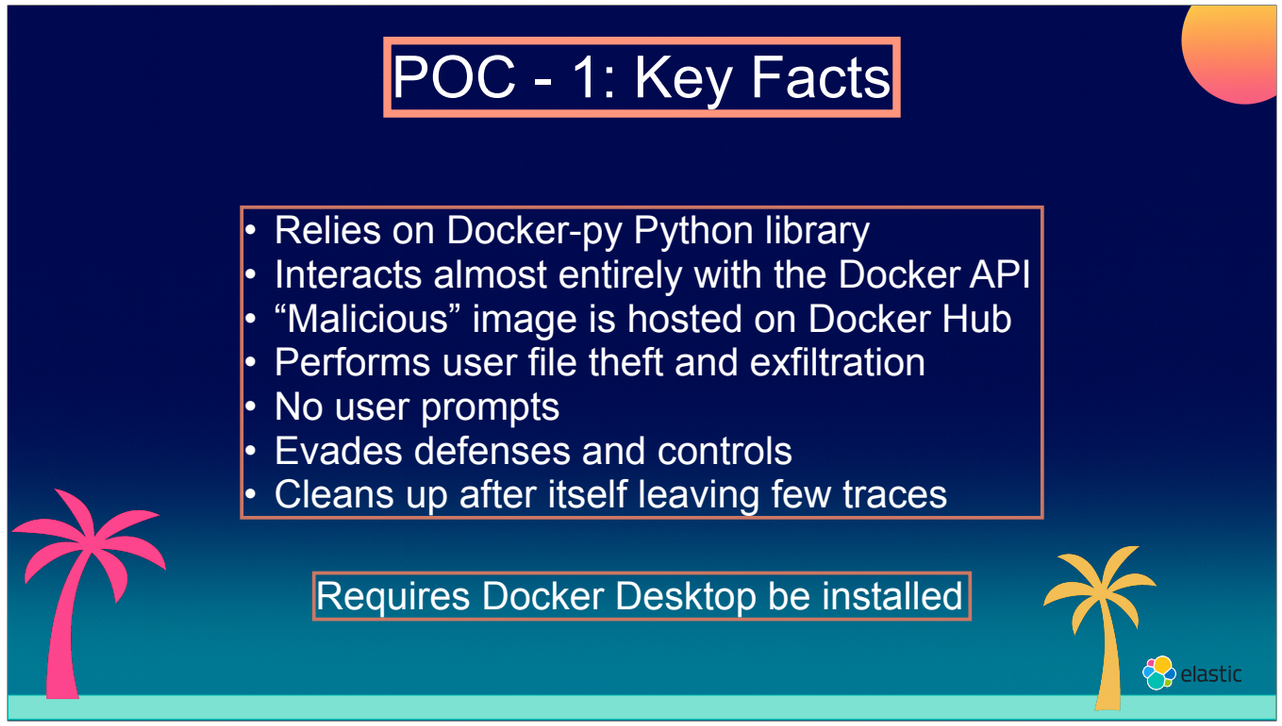
Like I just said the first POC is built with Python. It will abuse Docker settings and APIs in order to carry out its malicious goals.



For those that are not aware Docker Engine exposes a versioned RESTful HTTP API (the Docker Engine API) that clients use to control the daemon.

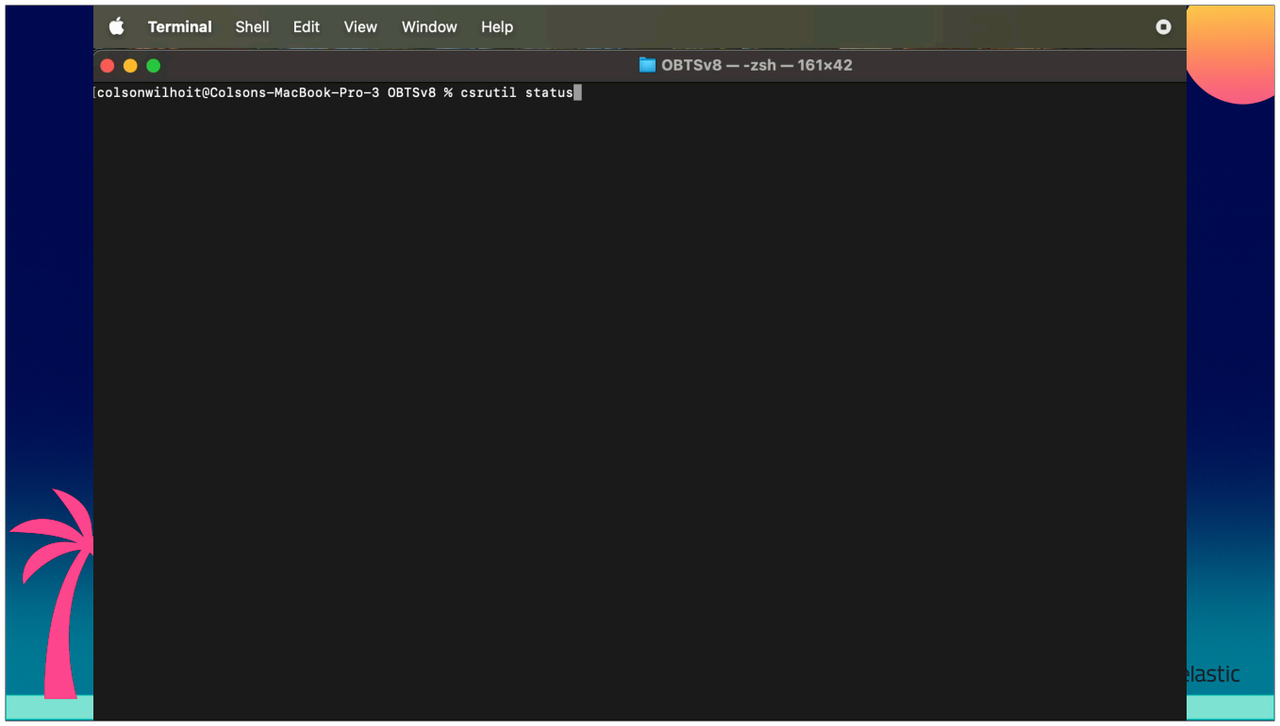
It’s exposed locally via a Unix domain socket /var/run/docker.sock

The CLI and libraries, like these and others, simply wrap these calls and speak HTTP to the socket



Some key facts to note:
This POC relies on the Docker-py Python library
It interacts almost entirely with the Docker API and nothing else
Container image is hosted on Docker Hub in private repo
Its built to be an information stealer, steal “secret” files, keychain, ssh keys, whatever and exfiltrate them from the system but this payload could do anything from dropping and executing a file on host to running recon commands and more
Its very fast, produces no user prompts and sidesteps defenses and controls
It even cleans up after itself leaving almost no trace it was ever there

Only prerequisite is that Docker Desktop be installed, doesn’t need to be running or anything



Ok here we go. Lets download and use a container to steal some secrets and exfiltrate them from my machine.

PLAY DEMO

Hopefully it wasn’t too fast and you saw but SIP was enabled and we were not root but thats it, the whole attack took seconds and there is almost nothing to indicate it ever happened and it would be way faster if I wasn’t running checks and outputting everything to the terminal.

```
colsonwilhoit@Colsons-MacBook-Pro-3 OBTSv8 % python3 obtsv8_docker_runner.py
[i] Docker socket not found at /var/run/docker.sock
[i] EnableDefaultDockerSocket is disabled, enabling it...
[✓] Enabled EnableDefaultDockerSocket in Docker settings
[i] Socket setting was changed, Docker Desktop needs to be restarted
[i] Stopping Docker Desktop to apply socket setting changes...
[i] No Docker Desktop processes found to stop
[i] Docker privileged helper not running
[!] The privileged helper daemon is required to create the Docker socket symlink
[!] You may need to:
[!]   1. Restart your Mac
[!]   2. Or reinstall Docker Desktop
[!]   3. Or run: sudo launchctl load -w /Library/LaunchDaemons/com.docker.socket.plist
[i] Docker socket service is not loaded
[i] Docker daemon not reachable via /var/run/docker.sock. Applying headless start logic…
[+] Set OpenUIOnStartupDisabled=true in /Users/colsonwilhoit/Library/Group Containers/group.com.docker/settings-store.json
[i] Starting Docker Desktop (headless)…
[i] Waiting for Docker daemon (timeout 180s)…
[✓] Docker daemon is reachable.
[i] Pulling image: colsonwilhoit/obts:v15.0 …
[✓] Pulled colsonwilhoit/obts:v15.0 (ID: sha256:77878758d3f8608c08ab840dc5349e27e6085866cd8fbaa4a5873a735a179d2f)
[i] Starting container…
[✓] Container started: obts_runner (b5b10cec0ab0bb1e26aff3c04bdf4b43de1e060ac7f4d98f25031648c7740865)
[i] Waiting for container obts_runner to exit…
[i] Container exited with status: 0
[✓] File upload verified: Download URL: https://tmpfiles.org/dl/1759642/secrets.zip
[i] Removing container obts_runner…
[✓] Removed container obts_runner.
[i] Removing image sha256:77878758d3f8608c08ab840dc5349e27e6085866cd8fbaa4a5873a735a179d2f…
[✓] Removed image sha256:77878758d3f8608c08ab840dc5349e27e6085866cd8fbaa4a5873a735a179d2f.
[✓] Cleanup complete. Ready for the next run.
[i] Restoring OpenUIOnStartupDisabled to original value (false) in settings-store.json...
[✓] Restored OpenUIOnStartupDisabled=false in /Users/colsonwilhoit/Library/Group Containers/group.com.docker/settings-store.json
[i] Restoring EnableDefaultDockerSocket to original value (false)...
[✓] Restored EnableDefaultDockerSocket=false
[i] Stopping Docker daemon since we started it...
[i] Sending termination signal to Docker process...
[✓] Docker Desktop stopped gracefully.
[i] Total execution time: 13.49 seconds
colsonwilhoit@Colsons-MacBook-Pro-3 OBTSv8 %
```

Breaking down what’s happening here we first check if the Docker daemon is present and running.

If its not we then check the Docker settings store to see if the DefaultDockerSocket is disabled and if it is we enable it

We then ensure we can start the Docker daemon without spawning the GUI by changing another setting value in Docker’s setting store.

We then start Docker headless wait for the daemon to become available then prompt it to pull our malicious container from Docker Hub.

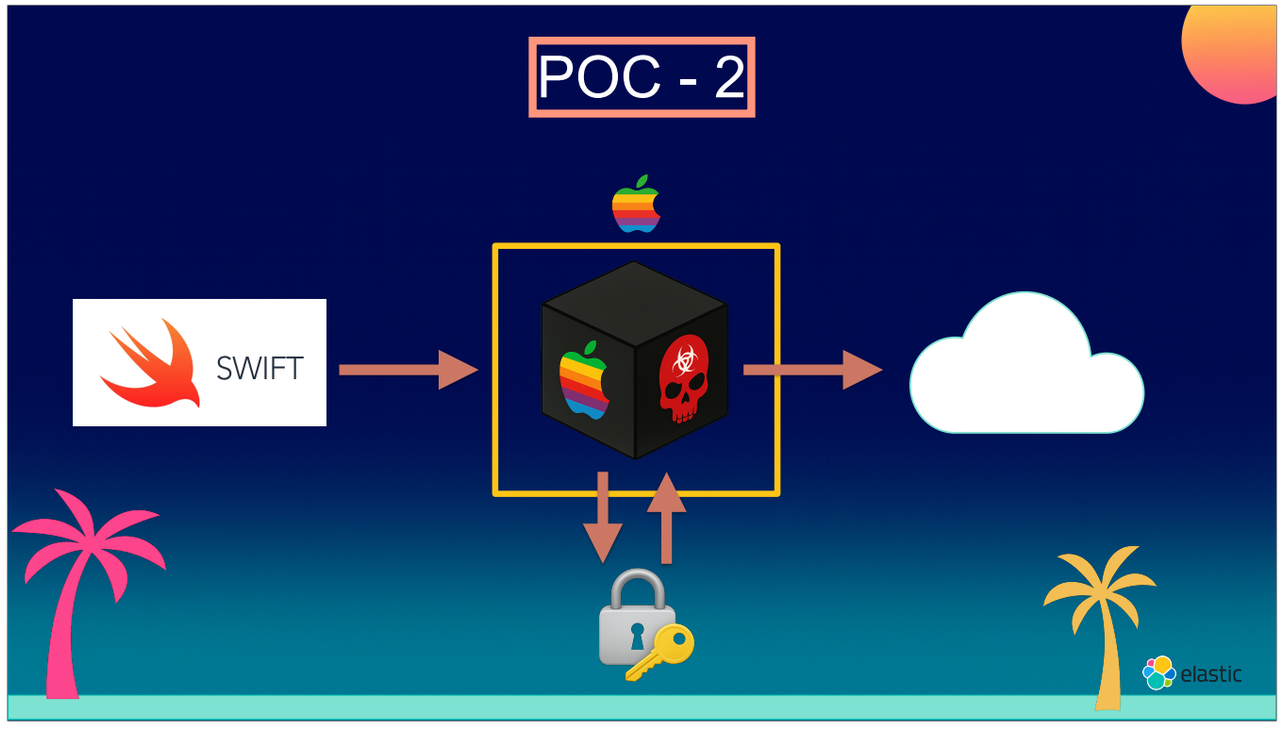
We then tell Docker to run our container mounting the host folders we specified to our container.

The container runs, executes its payload and exits.

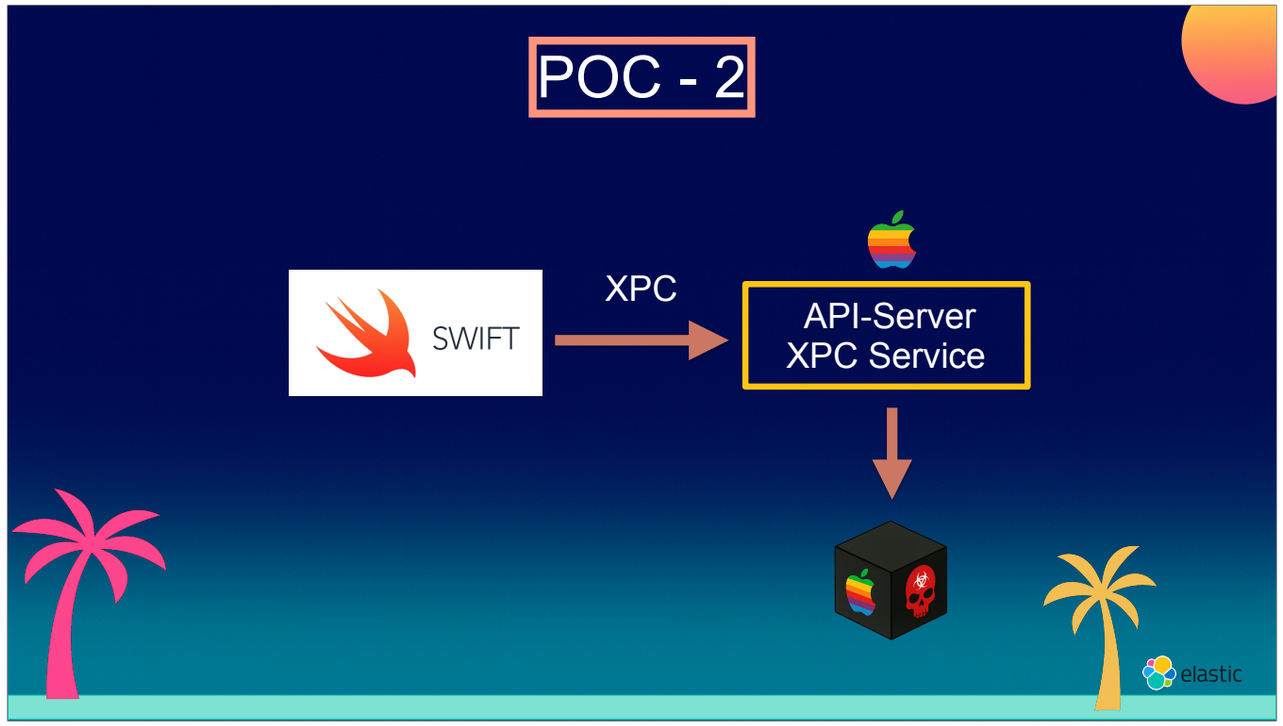
Successful data exfiltration is validated.

The container is then removed along with its image. The Docker settings file is reverted back to its original state and the Docker daemon is set back to its original state.

All of this in seconds without alerting the user.

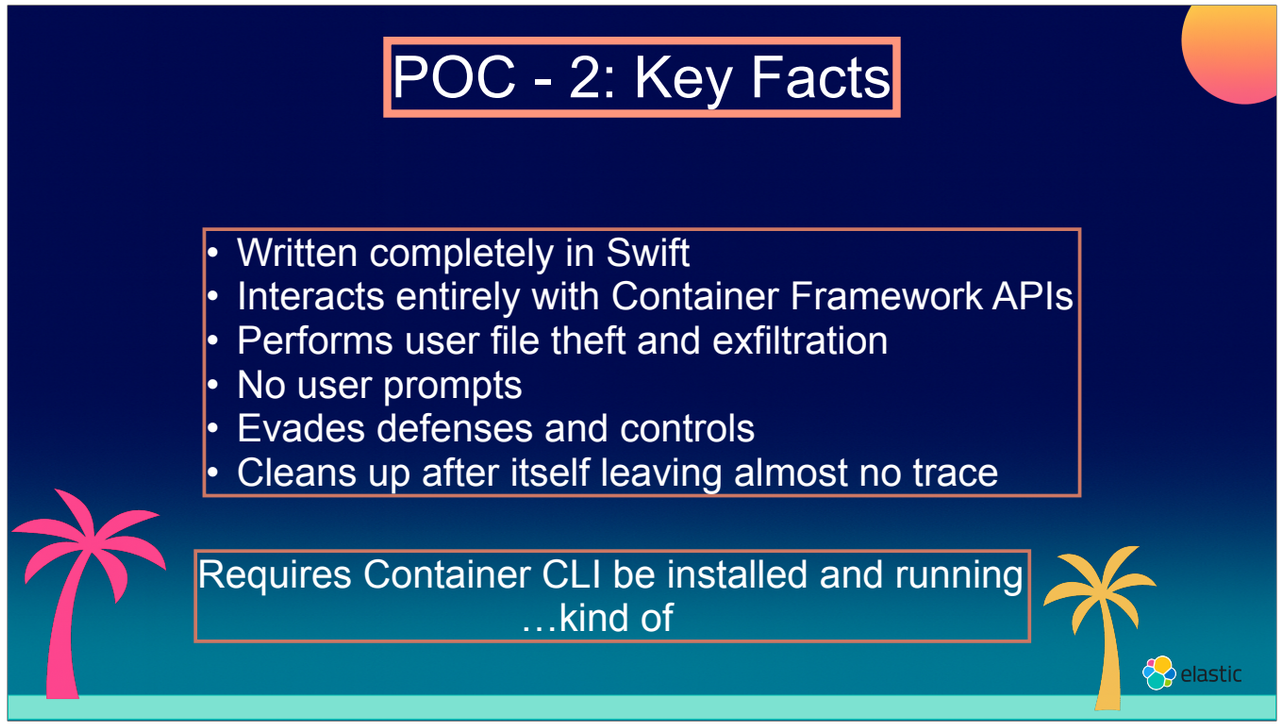


The second POC is built with Swift and will leverage Apple’s new Container Framework Client APIs in order to carry out its malicious goals.



In order to easily interact with the Container Client APIs you will need the Container CLI installed, which includes the api-server daemon that hosts the XPC service the API communicates over. You can get the CLI package from Apple’s Container Github.

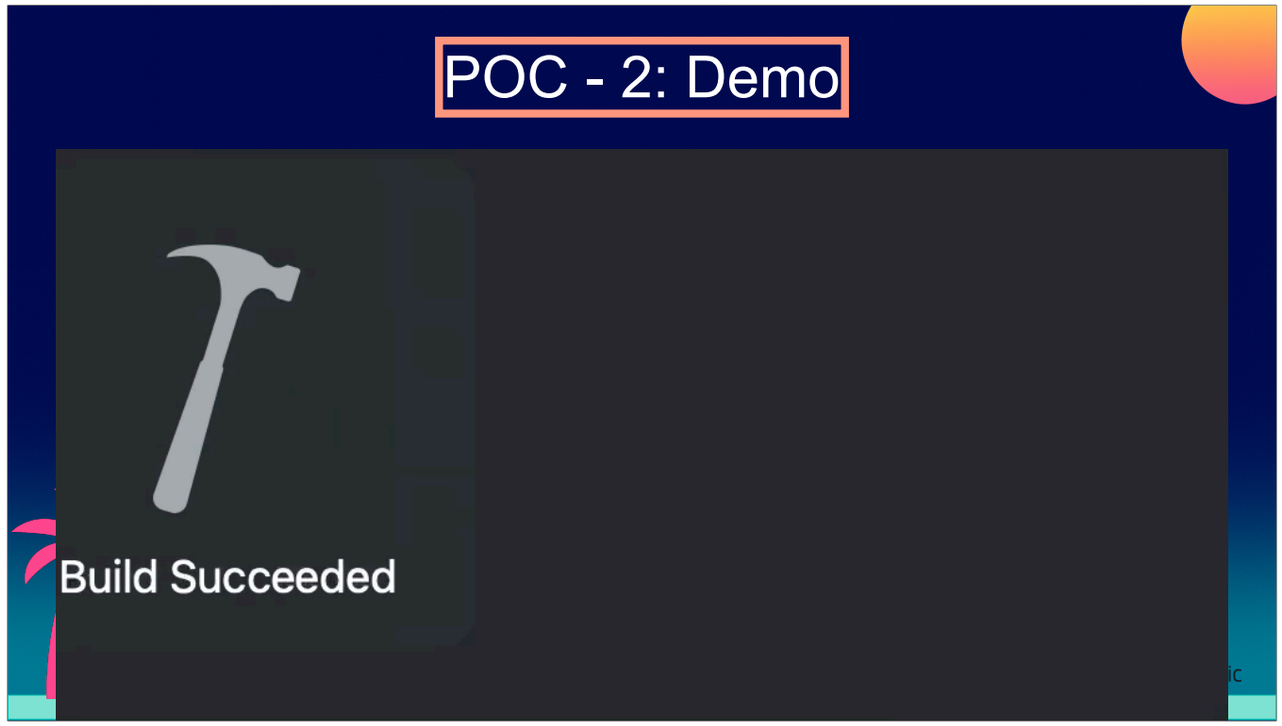
XPC for those unfamiliar is similar to RPC on Windows. An inter-process communication mechanism with a client-server architecture. The API provides calls we make to download images and import images, run containers, delete containers, etc.



Some key facts to note:
This POC is written completely in Swift and its less than 50 lines of code
It interacts entirely with the Container Framework APIs
Performs user file theft and exfiltration same as the last POC
Again produces no user prompts, evades all defenses and controls
Happens blazingly fast and cleans up after itself

A prerequisite for this POC is that the Container CLI be installed. Technically though you could build and link against the Container framework itself, which exists natively on Tahoe, which would not rely on having the CLI and api-server daemon installed but this would require a little more over-head in terms of code causing your binary to be a little larger but its possible.

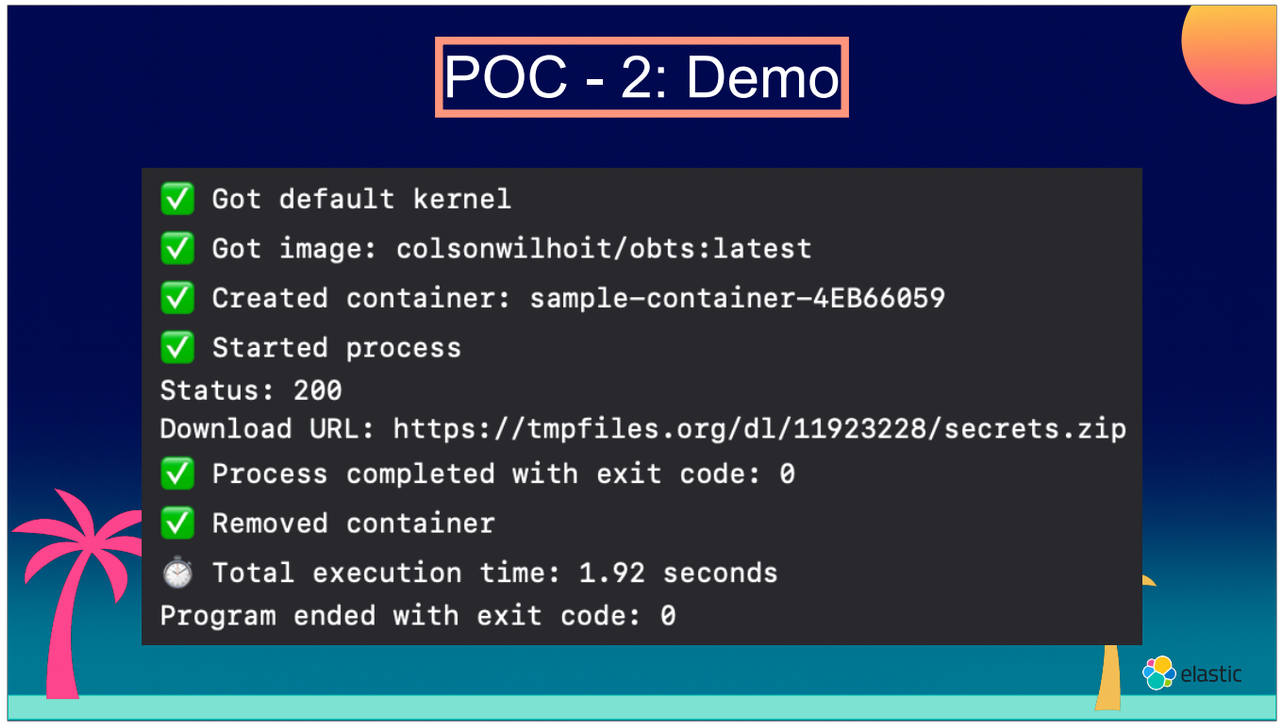
Another option is to bring the CLI binary along and prompt the users for their password installing it in the background. Out of for scope today though.



Alright here we go.

PLAY DEMO

Even quicker than the Docker POC. User secrets and credentials gone in a matter of seconds. Again SIP enabled and Gatekeeper as well.

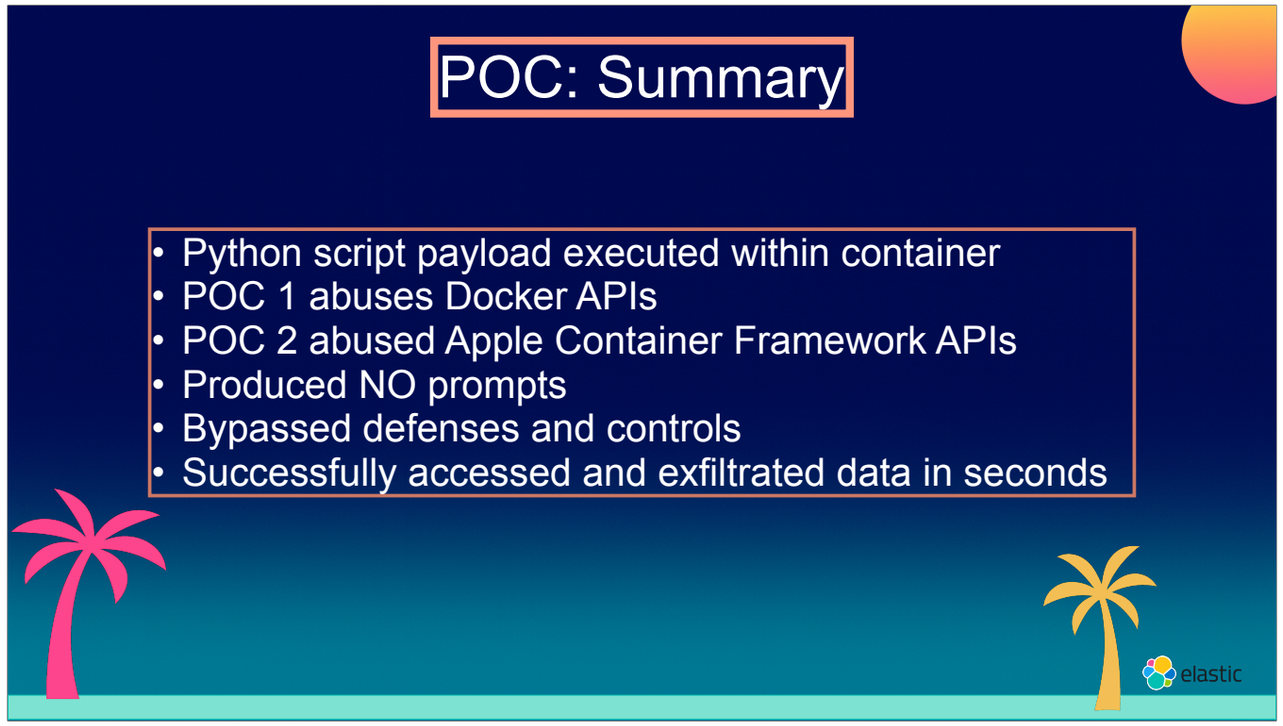


First we fetch the default kernel for the Linux VM our container will run inside of, supported by the current platform

Then pull our OCI (Open Container Initiative) compliant container image. The Container Framework only supports OCI image specification compliant images.

Configure our container to mount the host folders we want, user keychain in this case

Create and run the container, wait for it to exit and then remove it along with its cached image.

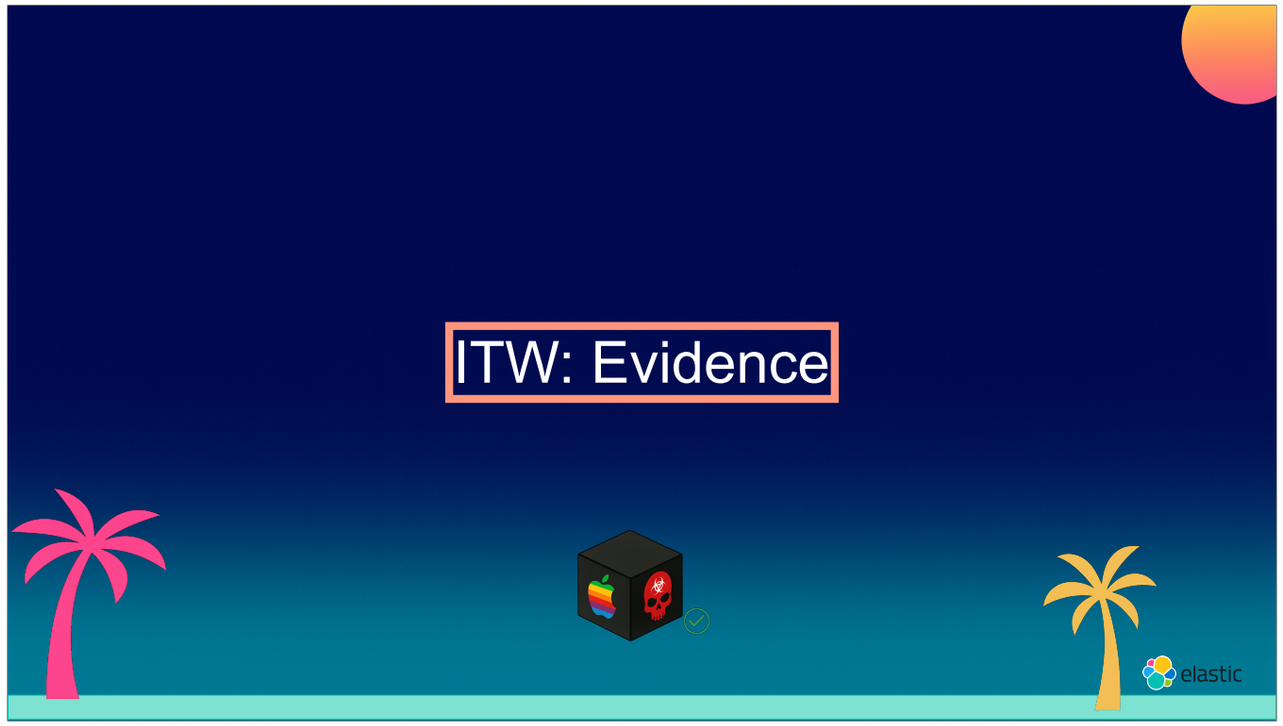


In summary, what I showed you was a very simple payload

Just a Python script that zipped up a folder and sent it out. Nothing fancy, nothing inherently malicious.

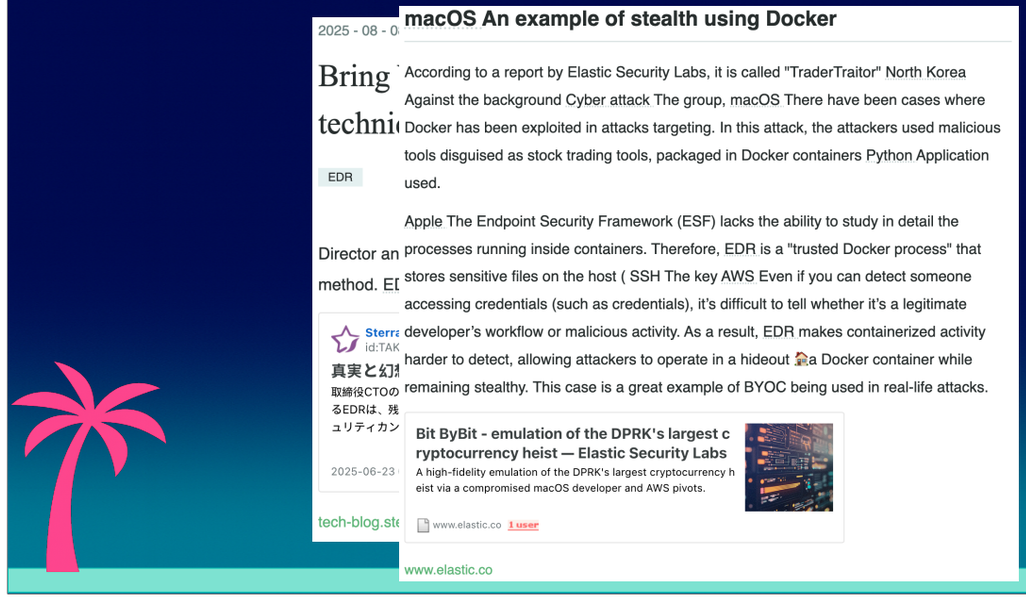
The real story is the container. By wrapping that payload script within a container image, I was able to download, run, and clean it up completely via APIs.

No prompts, no alerts, and all in seconds. That’s what makes this technique so effective.



Alright now that you’ve seen how it works lets look at some in the wild evidence about how this technique is being used by threat actors and red teams today.

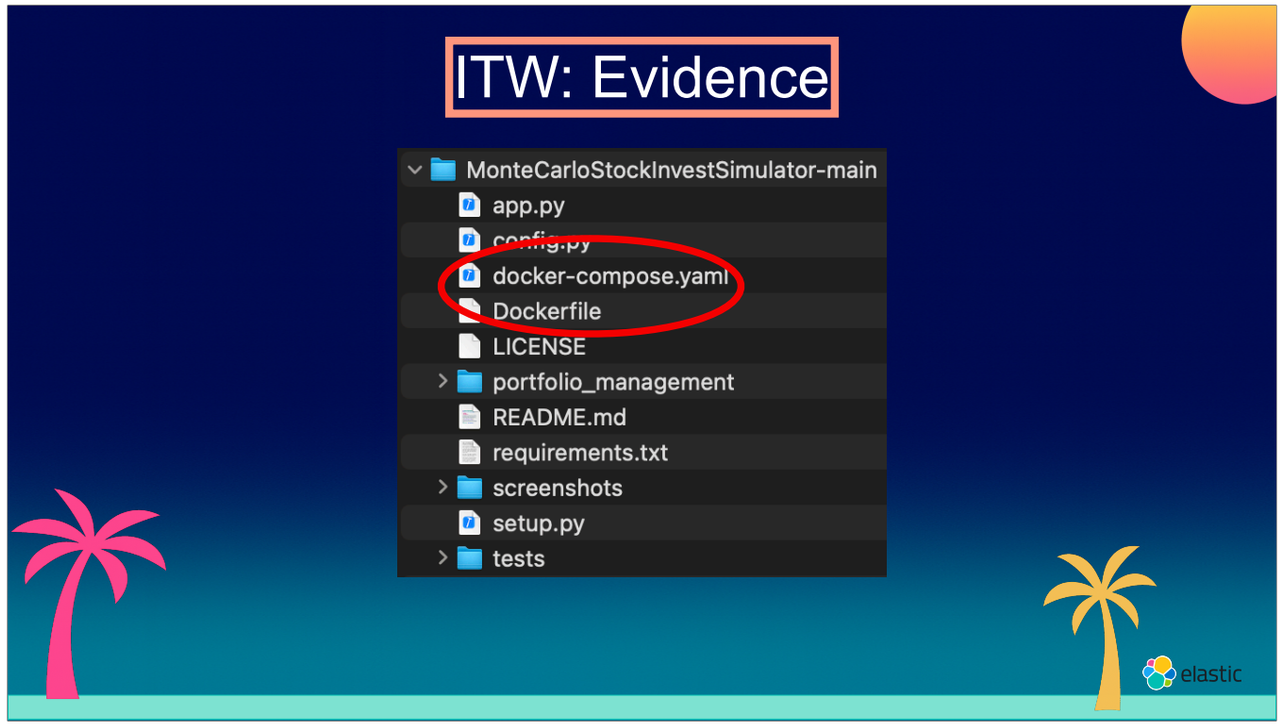
ITW: Evidence



Surprising to me the only research I could find, in existence, around this technique specifically, Isolated Defense Evasion, being used to bypass EDR was blogged about recently by a Japanese researcher, goes by tkmru, who actually turns out cited my own ByBit Compromise Emulation blog, from earlier this year, in his write-up.

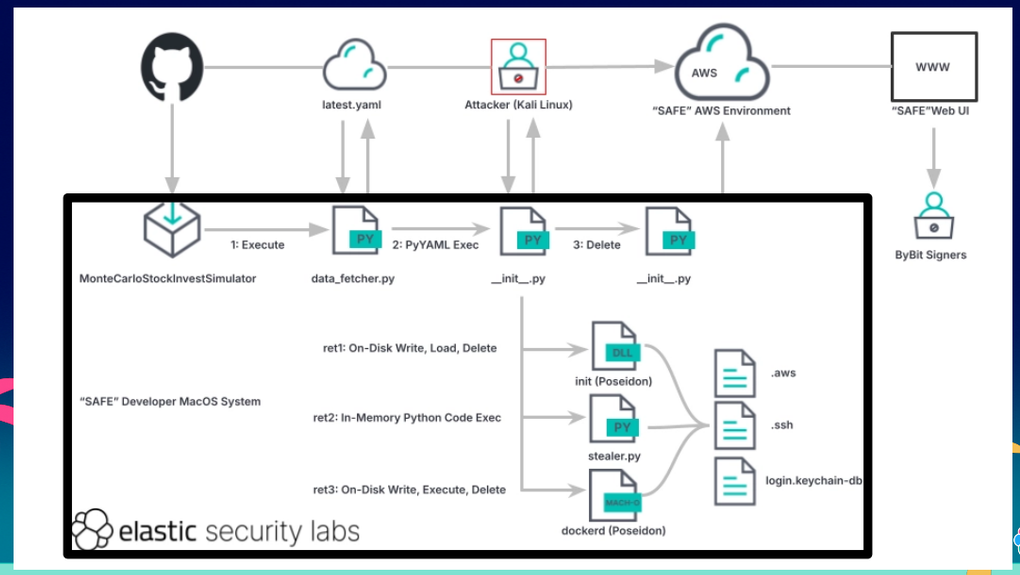


The first evidence I saw of this technique and how powerful it could be was actually when I was researching and emulating the ByBit compromise, if you all aren’t familiar ByBit is one of the largest crypto currency exchanges out there and hackers (most likely the DPRK Trade Traitor group) stole a record $1.5 billion worth of ETH from them by compromising their multi-signature wallet provider SAFE wallet. The intrusion began on a SAFE wallet developers MacOS system.



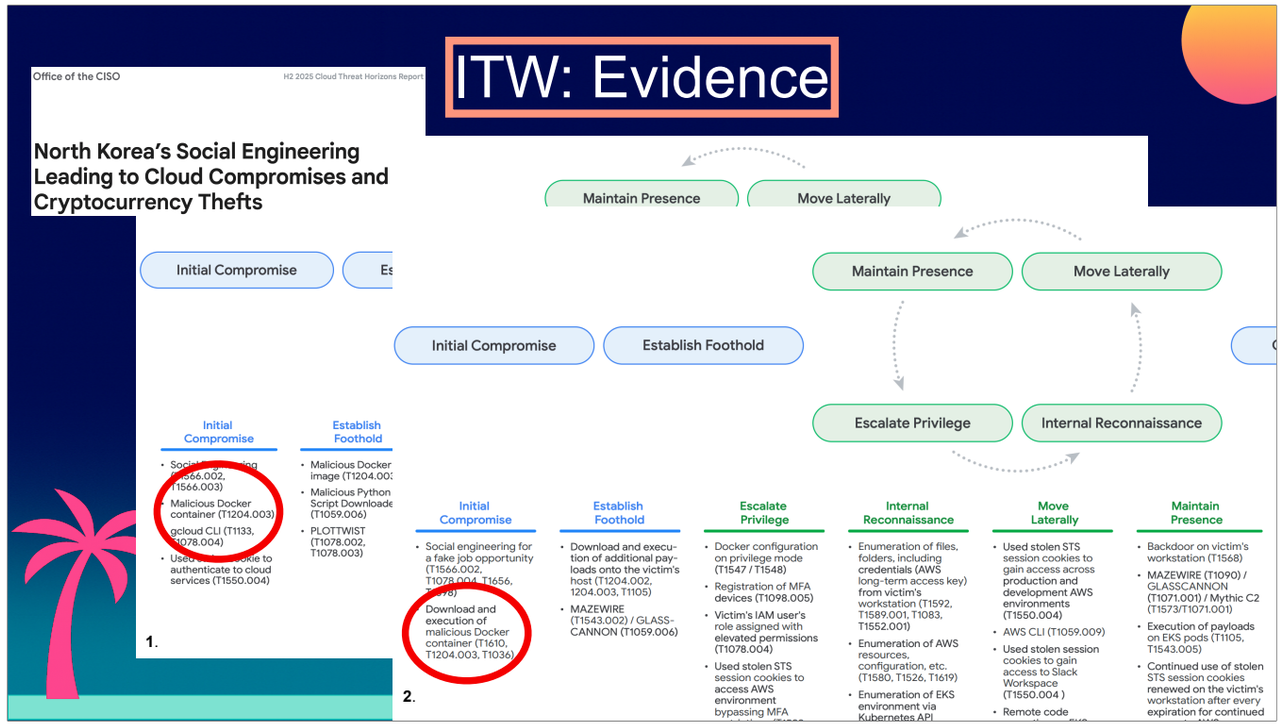
The initial access application that was used was Python based and presented the user with two execution options one of which was to run the Python app inside a Docker container. It was testing this Docker option that really made it hit home for me how powerful this could be when I saw how there was almost nothing to see. And I was like Uh-Oh.

ITW: Evidence



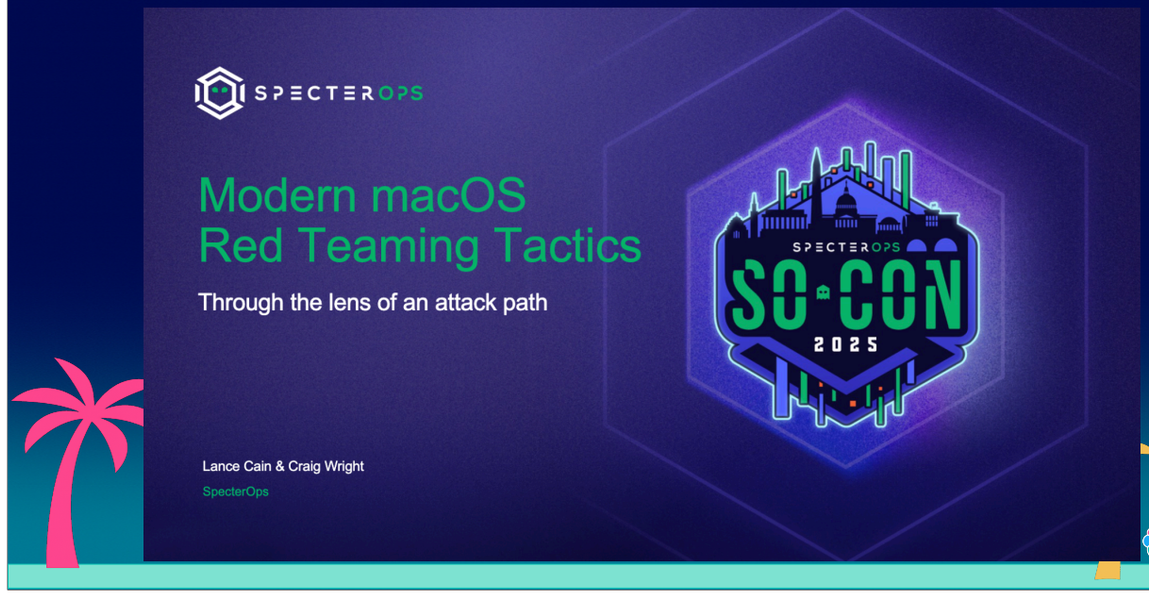
I won't go into all of the details regarding the Python application payload and how it works, you can go read the blog, but you can see everything the payload does here inside the black box which represents the container.

All of this, code exec, second-stage payload execution, credential harvesting, etc. can happen without the threat actor being worried about defenses or controls on MacOS.

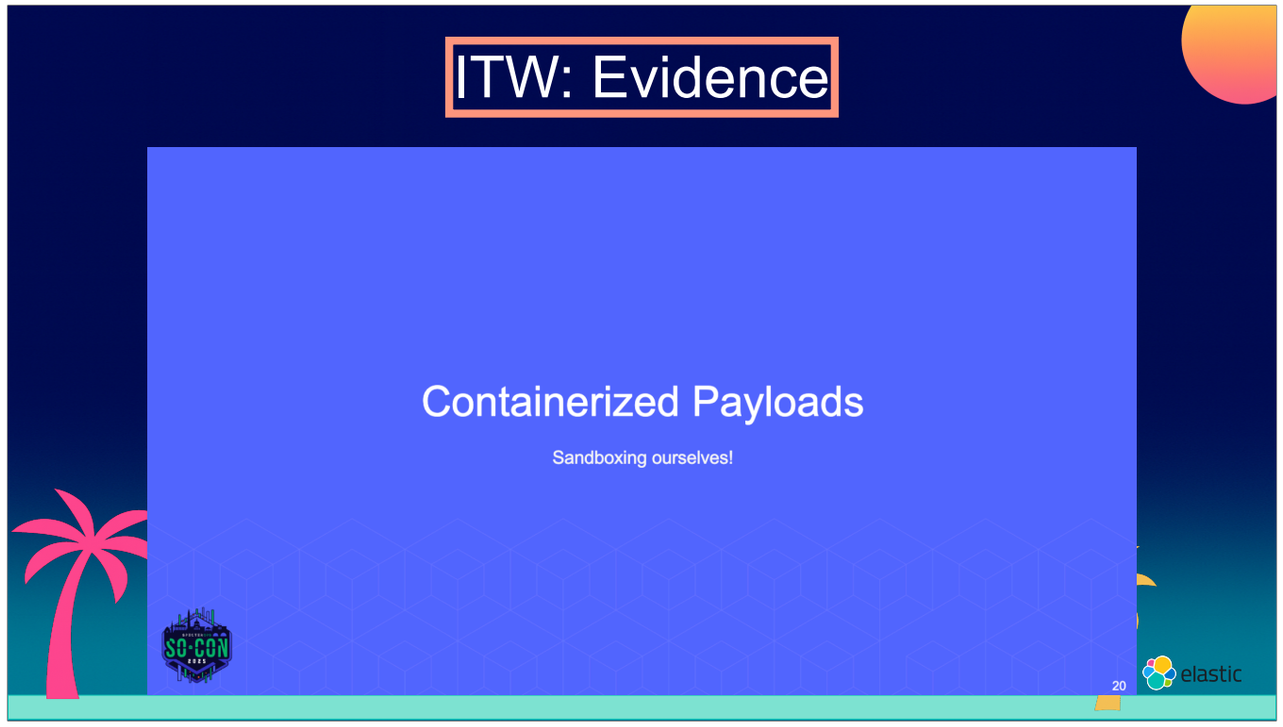


Recently Google released their H2 Cloud Threat Horizons Report in which Mandiant discussed recent DPRK campaigns, they have responded to. Both of the attacks they discuss involve the use of a container for payload execution. So threat actors are catching on and we should take note.

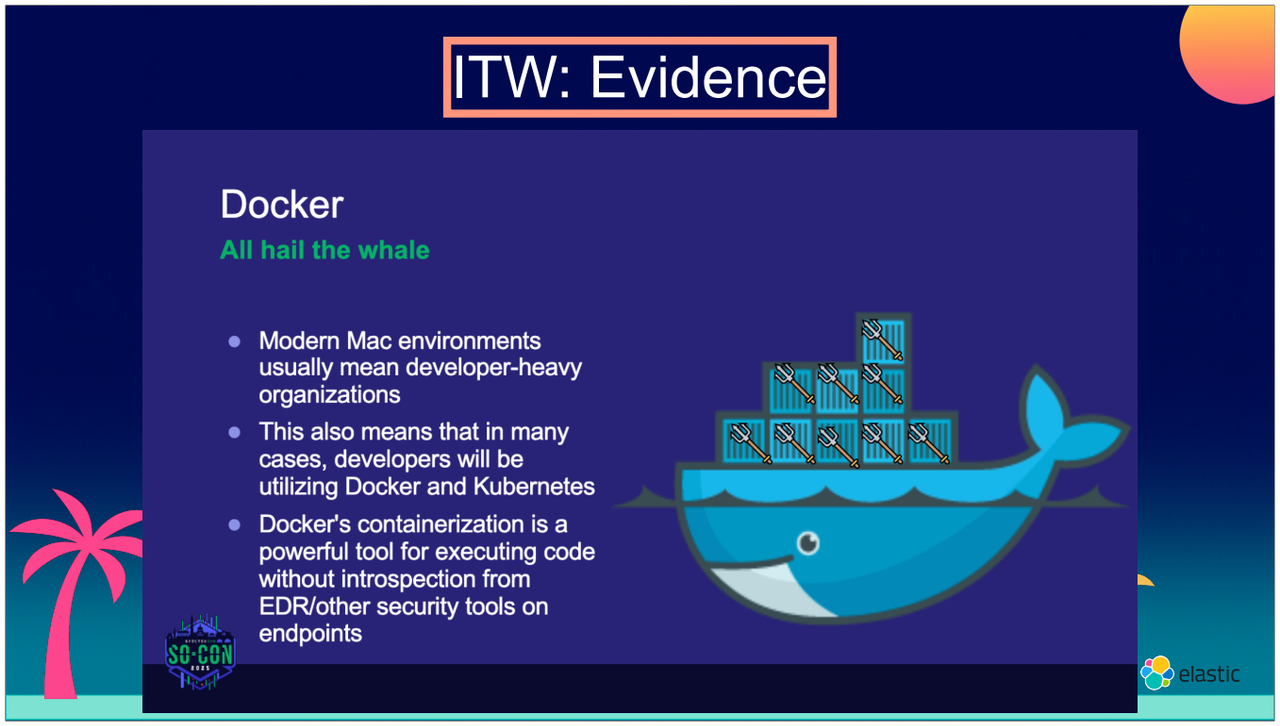
ITW: Evidence



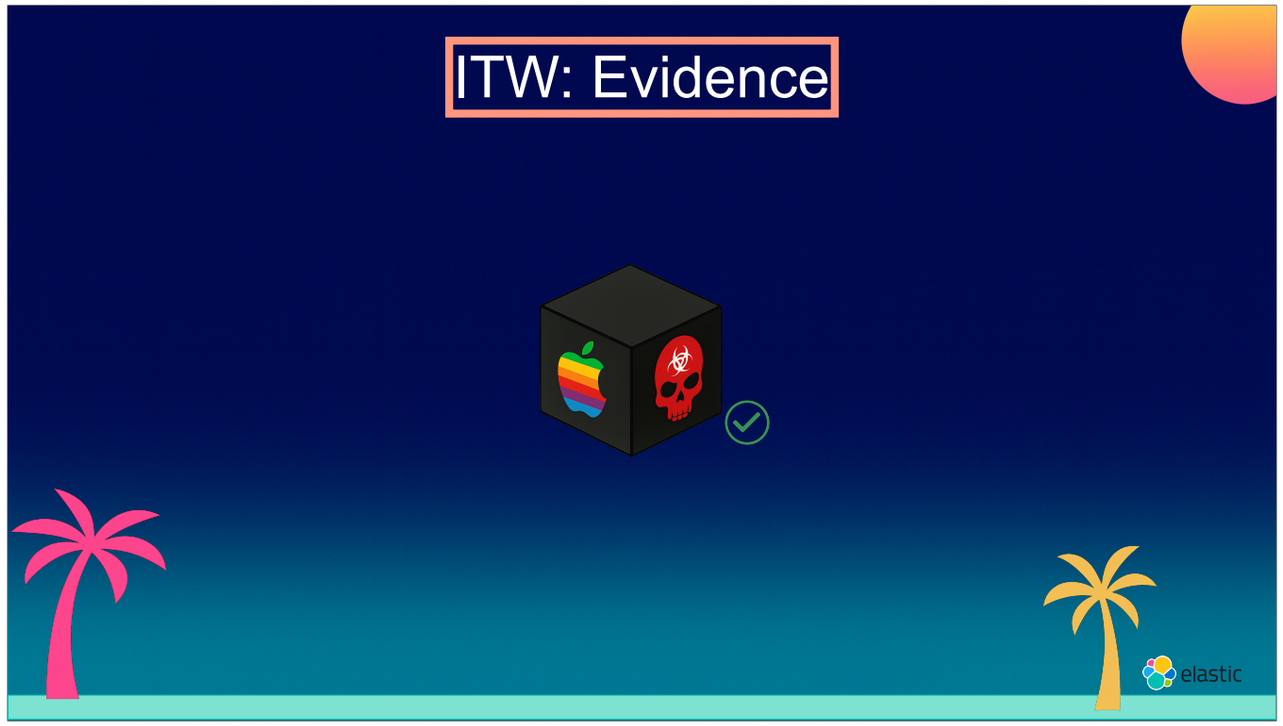
On the other side of the coin Red Teams are starting to talk about this technique as well. Many of you are familiar with SpecterOps, a great company, full of extremely talented engineers, operators and consultants. At their conference this year SO-CON two of their guys Lance Cain and Craig Wright gave a great presentation on modern macOS red teaming TTPs.



And one of the main techniques they discussed was containerized payloads!

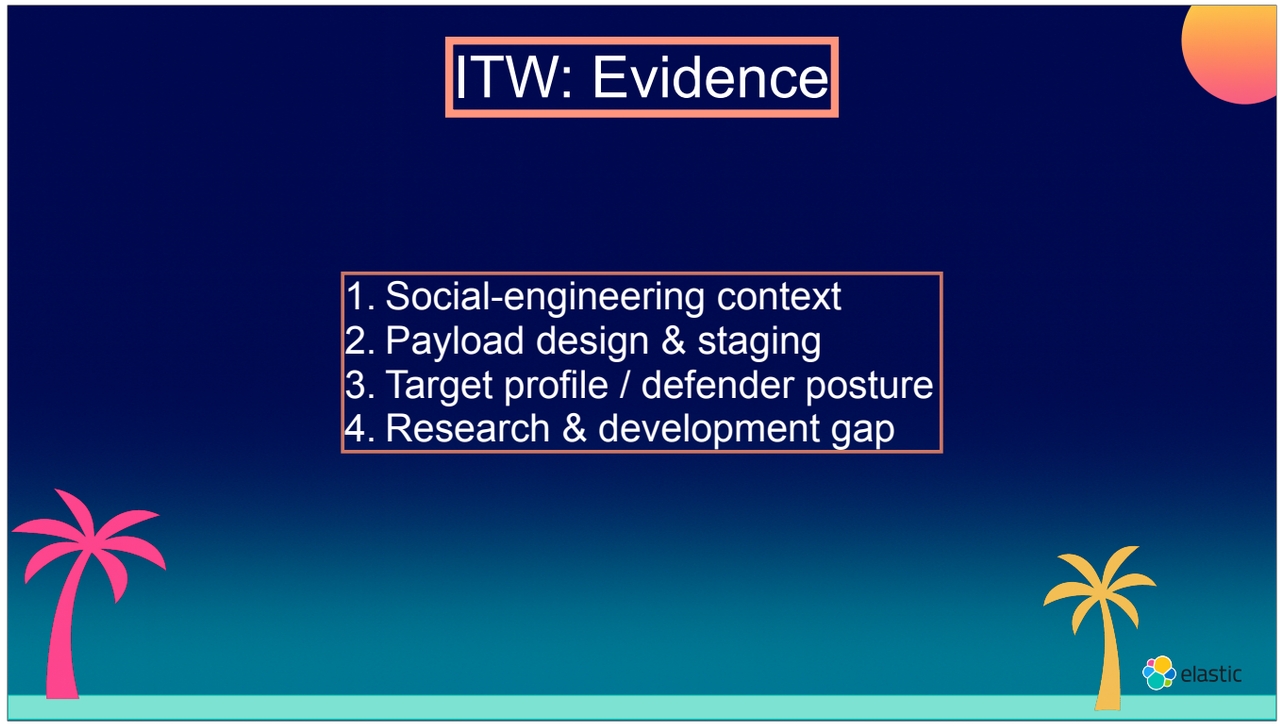


Using containers to shield themselves against the security controls macOS implements and nosy EDRs. So again its being utilized out there and gaining traction. Go check out this talk on YouTube if you haven’t already.



Now something to note with the DPRK and Red Team examples is that they are bringing their container payloads along with them as part of the initial access and rely on building their container on the victim system using an included Dockerfile as opposed to downloading the pre-built image and running it programmatically via APIs like my POCs did. Their containers were also more long-lived than mine which run and exit. Its an important difference to note.

There are some reasons though for the differences I find necessary to briefly mention



1. The lure determines what looks “normal” (e.g., a coding challenge or debug help project with an included Dockerfile makes sense for that context).

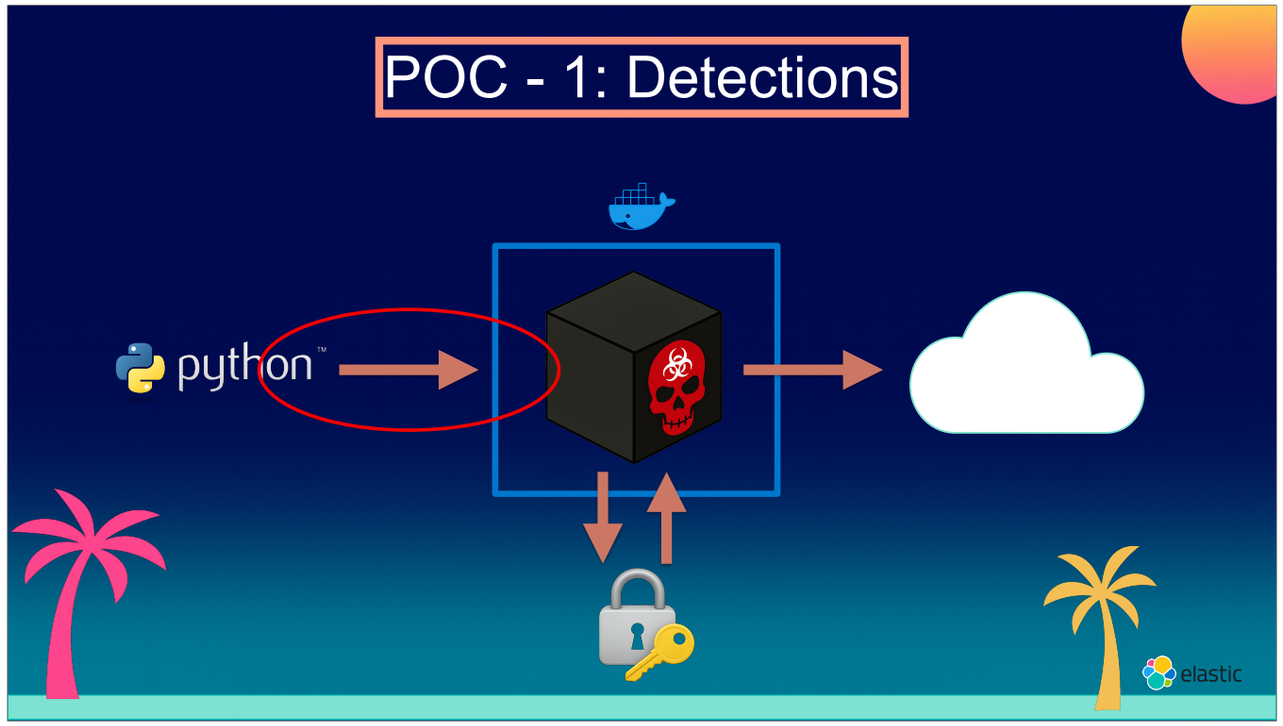
2. Payload type and staging needs (single image vs multi-stage) drive whether you embed build steps for the container or do what my POCs did.

3. If you expect lax controls (developer laptops, permissive networks), on-host builds are less of a concern; high-defense targets favor pre-built, programmatic flows. Threat actors are lazy and will take the necessary approach with the least amount of friction.

4. Pre-built-image + API approach I’ve presented today is currently much more OPSEC-clean and reliable; from what I’ve seen real operators haven’t adopted it yet but they will.



So how can we defend against and detect this? Can we even detect this? What do we need to do?



Lets tackle the first POC and see how we could defend against a threat actor abusing Docker to run their container payload

We will focus on the modification of Docker’s settings file and the abnormal execution of Docker itself.

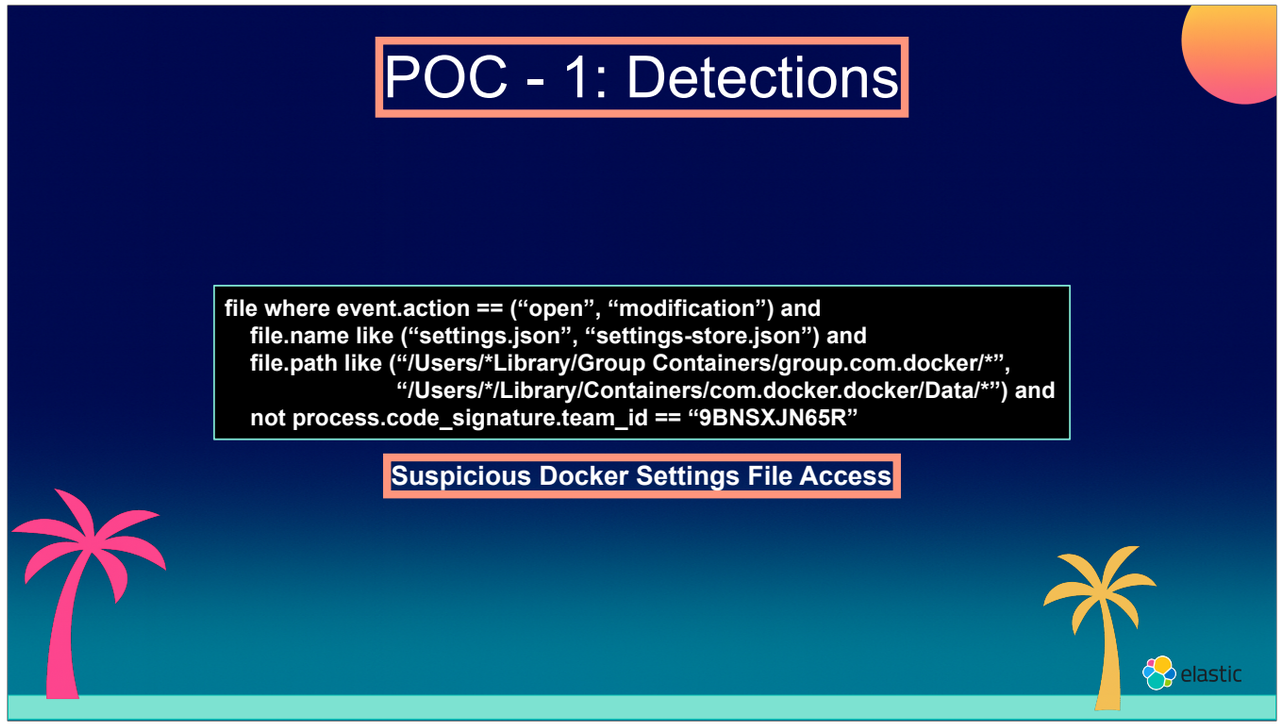
Of note there are other libraries, in Go and even Nodejs, allowing for interaction with the Docker API and of course a motivated and well funded threat actor could also create their own library to easily interact with Docker’s APIs.



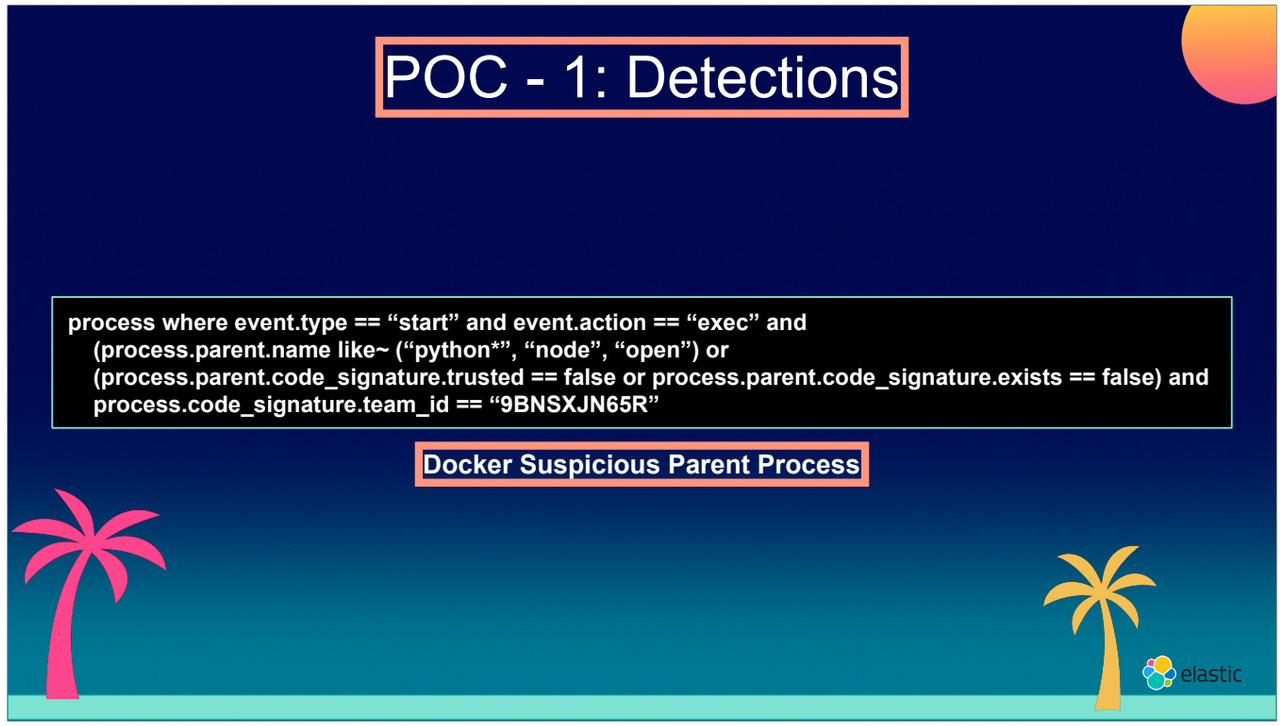
For those of you that don’t know Docker Desktop maintains its settings configuration in one of two files depending on your Mac OS and Docker version. These settings are actually writable allowing us to modify so called “safe” settings and restart Docker, in the background, to apply our changes.



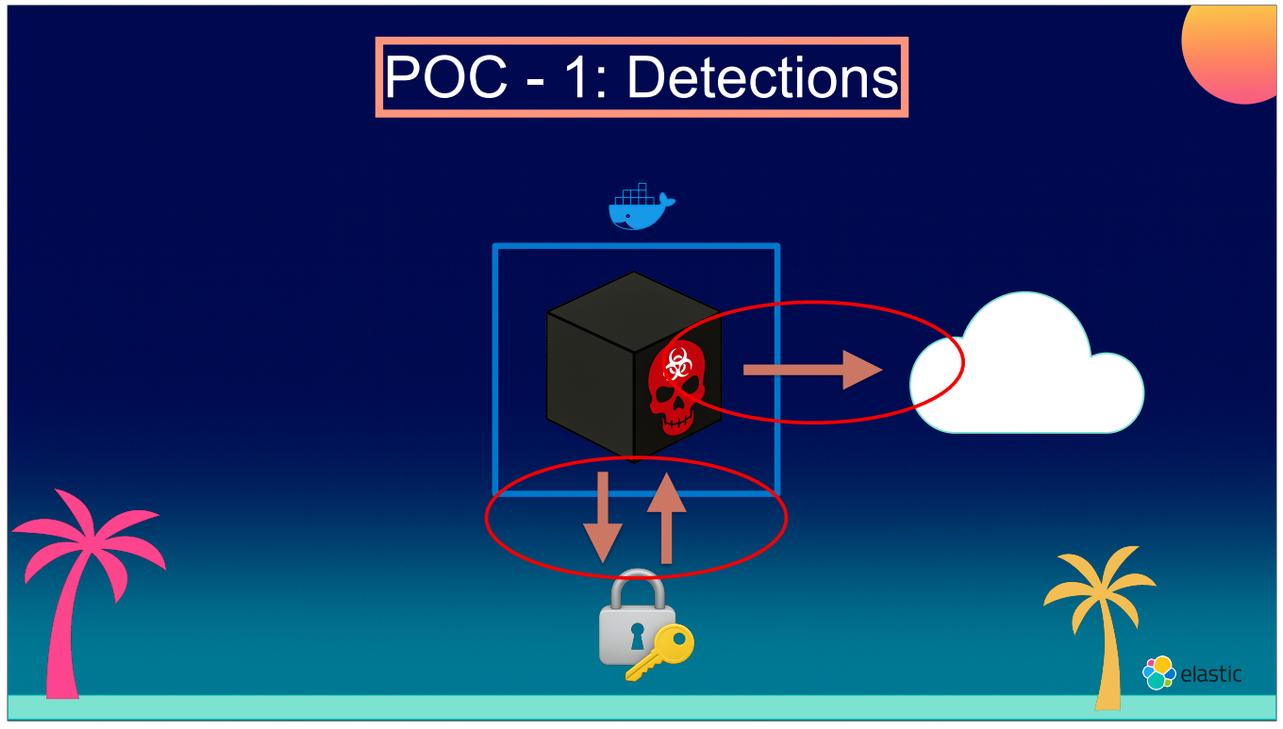
Ensuring those safe settings are enabled and implementing a tool like Santa by North Pole Security, that allows for File-Access Authorization, could allow us to lock down that Docker Settings File, and would prevent a threat actor from being able to abuse Docker in this way.



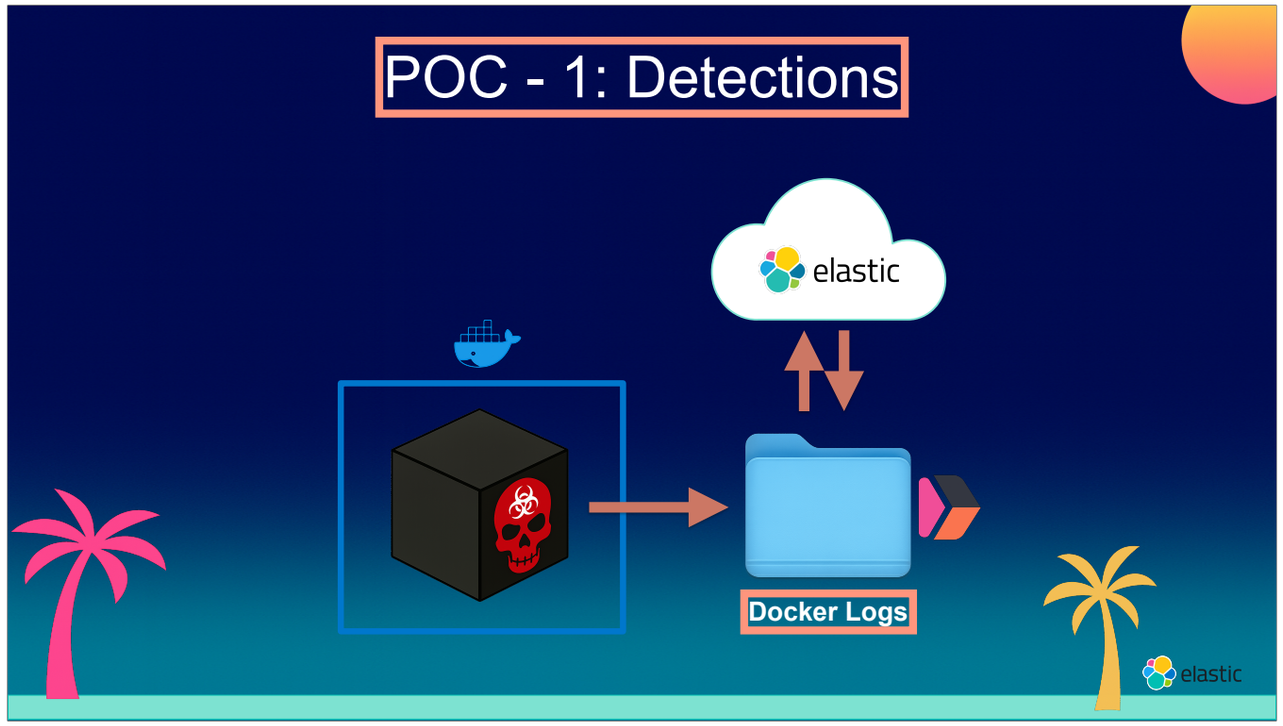
With Elastic I can build a detection rule that notifies me when this settings file is opened or modified by a process that is not a Docker process and if this rule was implemented as one of our Endpoint Behavior rules I could tie a response action to the rule that kills the process or responsible process that attempts this.



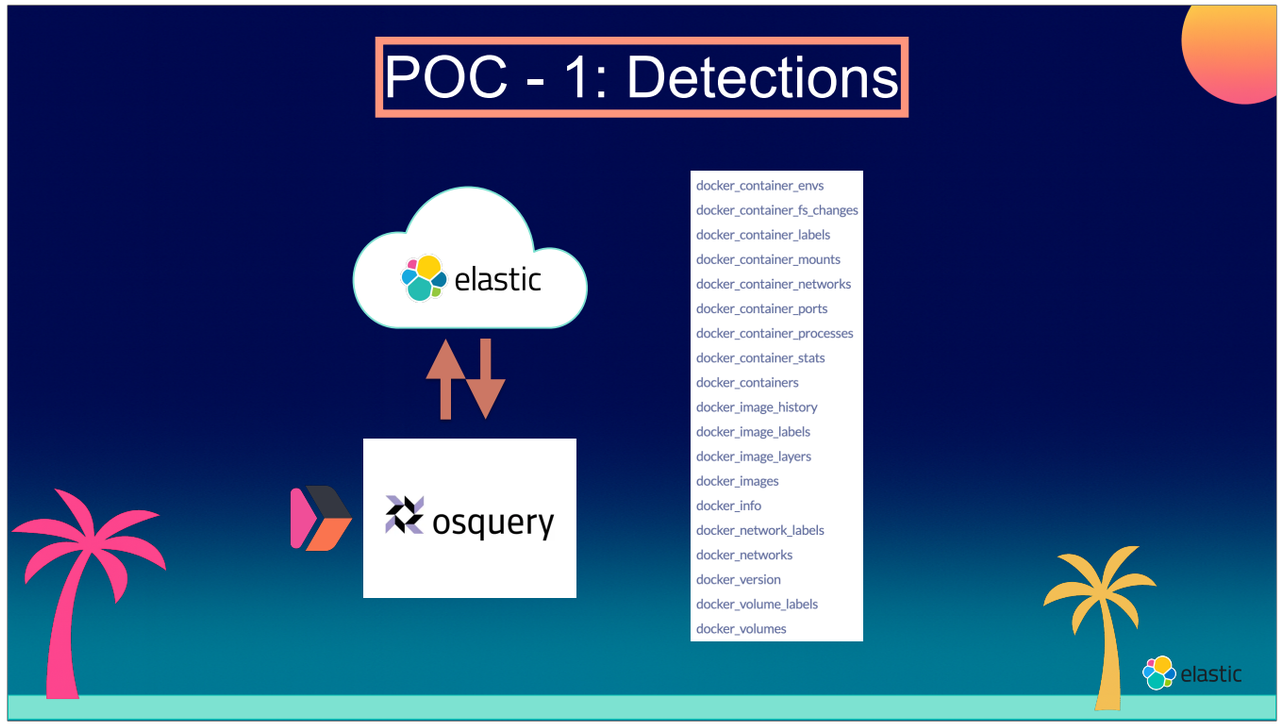
We can also create a detection rule looking for suspicious or abnormal executions of Docker. 99% of the time Docker Desktop will be executed via the GUI (double clicking) or on system start both of which result in the Launch Daemon spawning Docker Desktop so this is a pretty good indicator of something strange or investigation worthy. This detection could be bypassed by using the LSOPEN call or an XPC submit routine to execute Docker but its still a good detection to have in place.



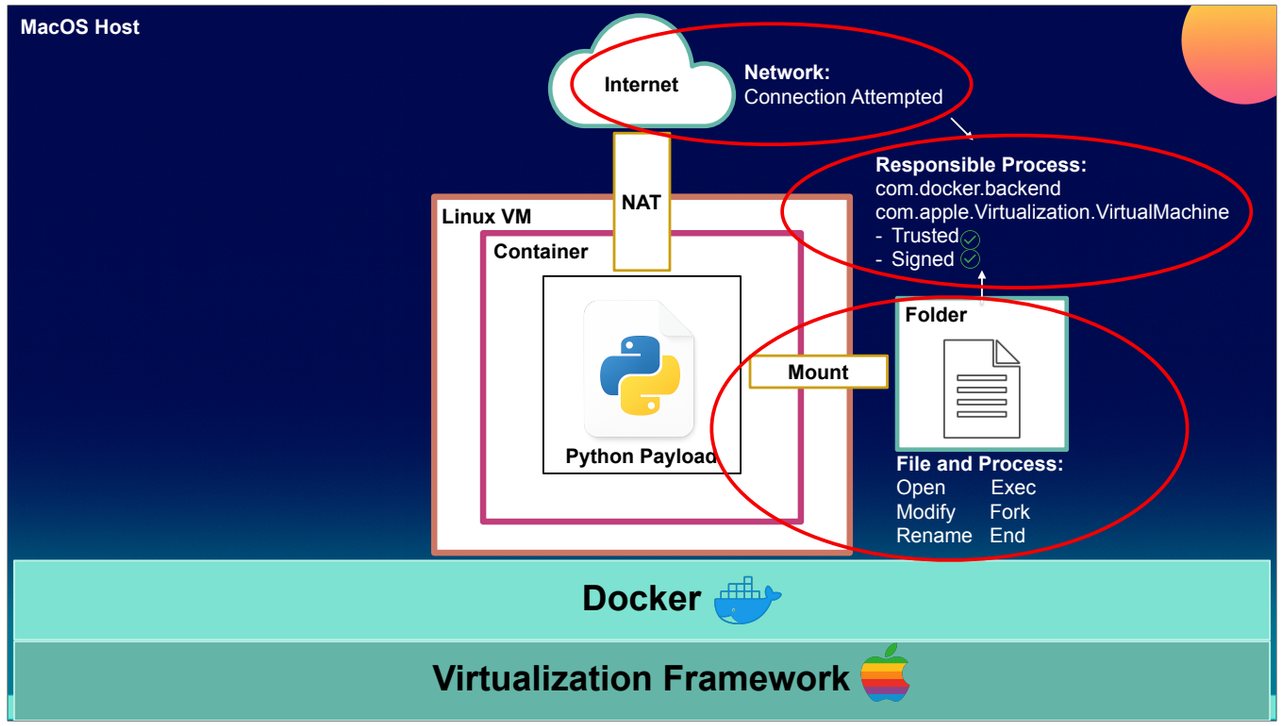
We can create detections for Docker sensitive file access and outbound network connections to suspicious top level domains but as I mentioned earlier from what I’ve observed these activities can be very noisy and detections here will need to be evaluated on a case by case basis alongside a baseline of allowed or normal activity.



Docker generates container logs that can provide valuable information regarding container images and container execution. Our Elastic Agent has a data integration that collects those and ships them back to your cluster where you can build detections around abnormal or suspicious usage.



Elastic Agent also has an OSQuery integration that allow you to run scheduled or live queries remotely from Kibana against your endpoints and OSQuery has a number of helpful Docker tables to pull from.

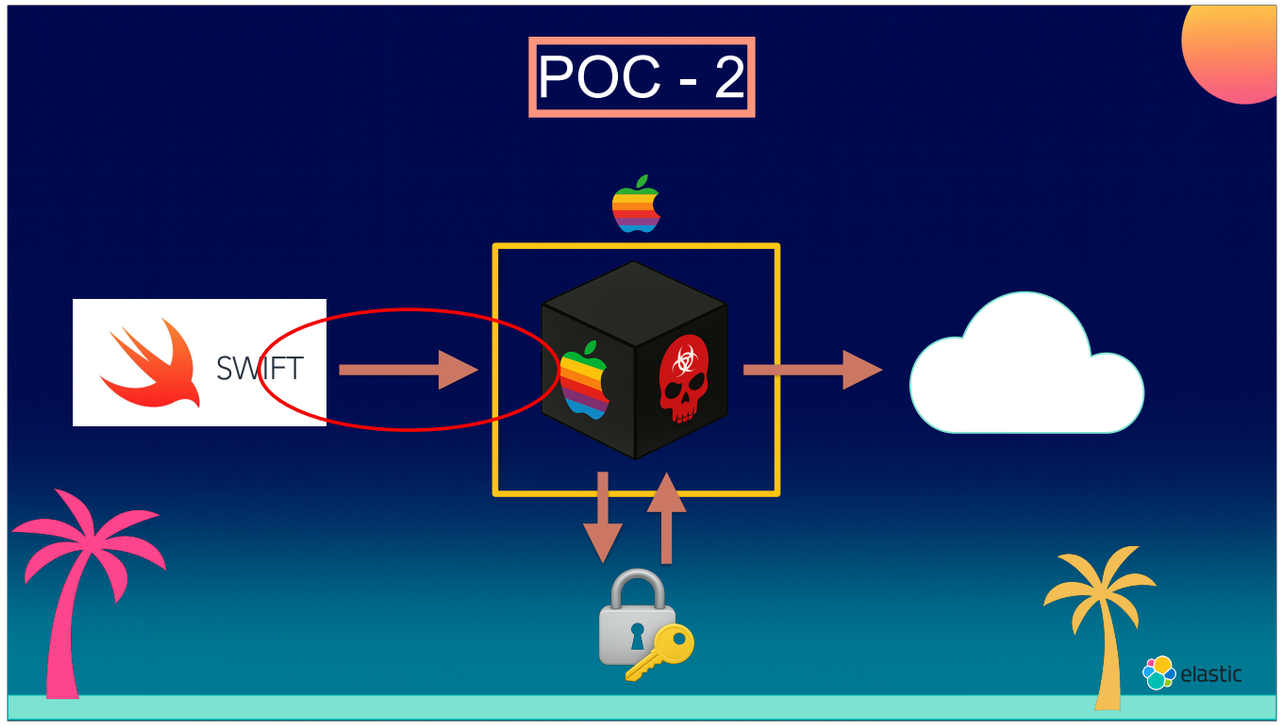


Lets make this perfectly clear though in regards to visibility and the real issue here which makes this technique so effective.

With an EDR and/or Endpoint Security Framework events we can see network connections outbound/inbound, we can see sensitive file open, modify, rename events and process executions, forks, etc. BUT

all of these behaviors are happening on behalf of the container by the Docker process com.docker.backend and the Apple Framework XPC Service com.apple.Virtualization.VirtualMachine, which are signed, trusted, platform binaries and they can be very noisy with no link to a specific container or host executor.

Thats the real power here is unless you have this activity baselined or locked down knowing if something is malicious or not is nearly impossible.



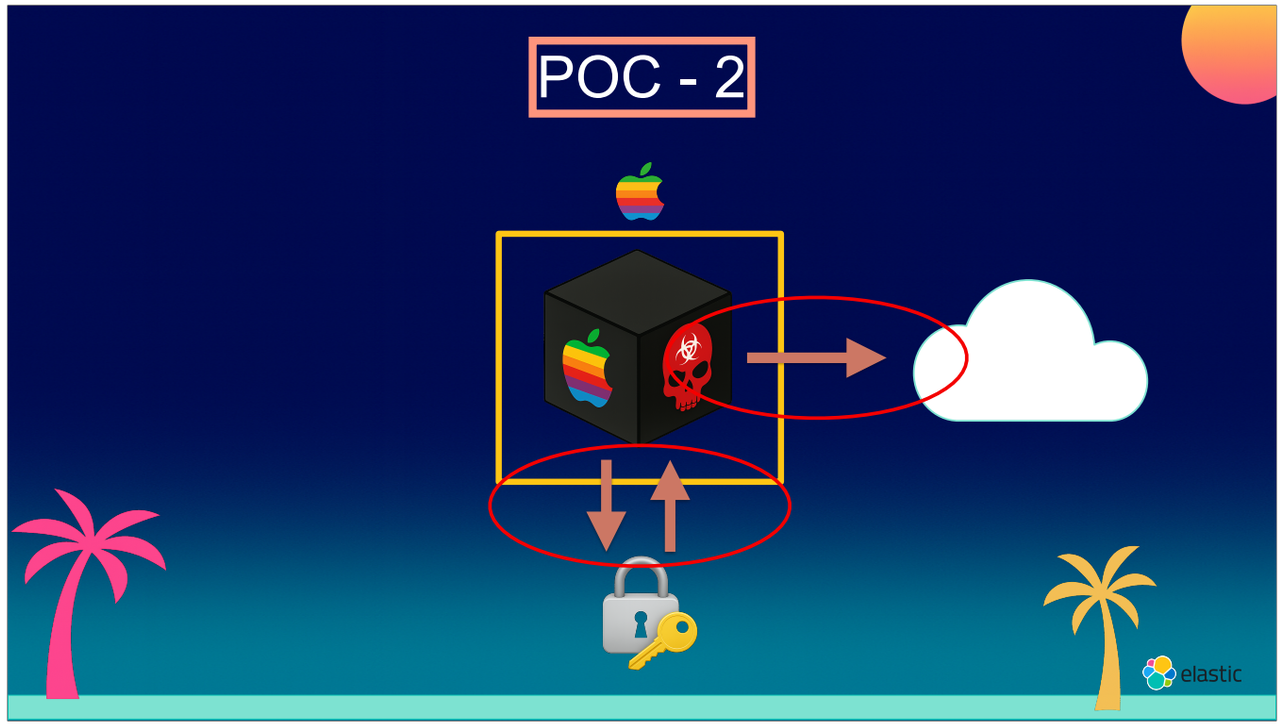
The second POC was built with Swift and used Apple’s new Container Framework APIs

The primary detection point here is the Swift binaries communication with the Container Framework services, as I showed earlier happens over XPC.

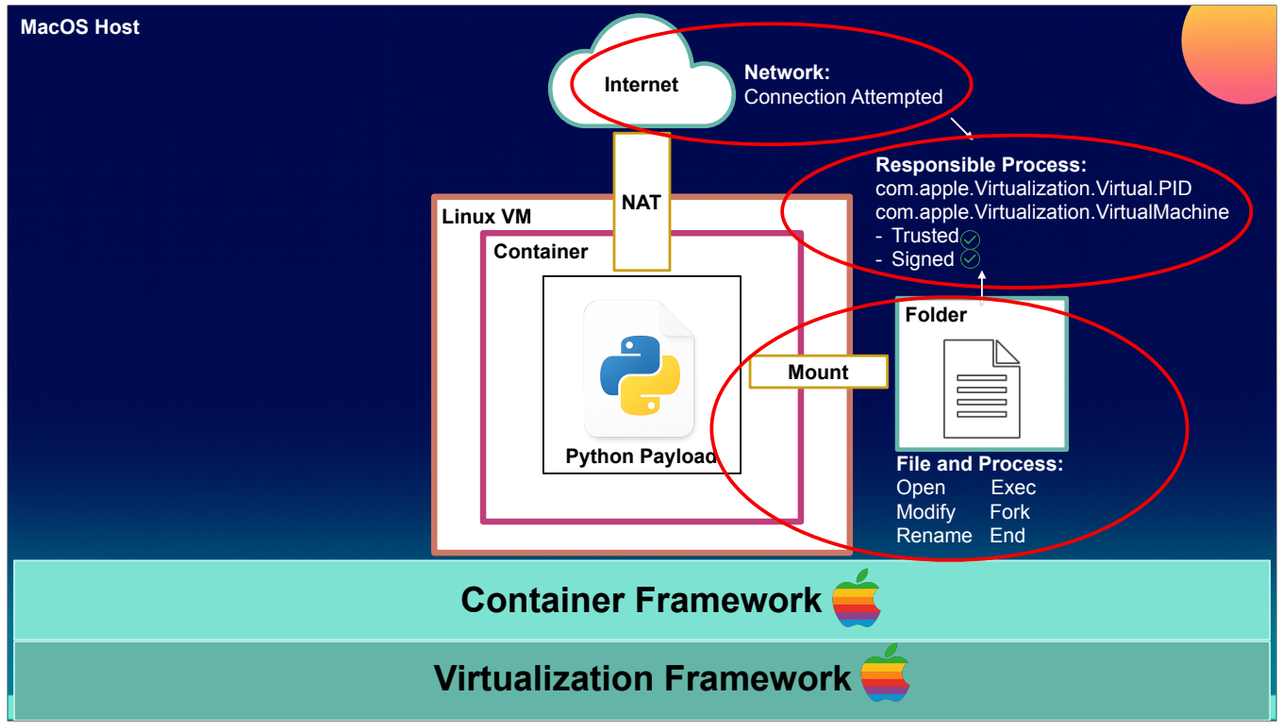
The Endpoint Security Framework collects XPC_Connect events that allow us to see the Swift binary talking to the Container XPC services in order import our image, run it and clean up.



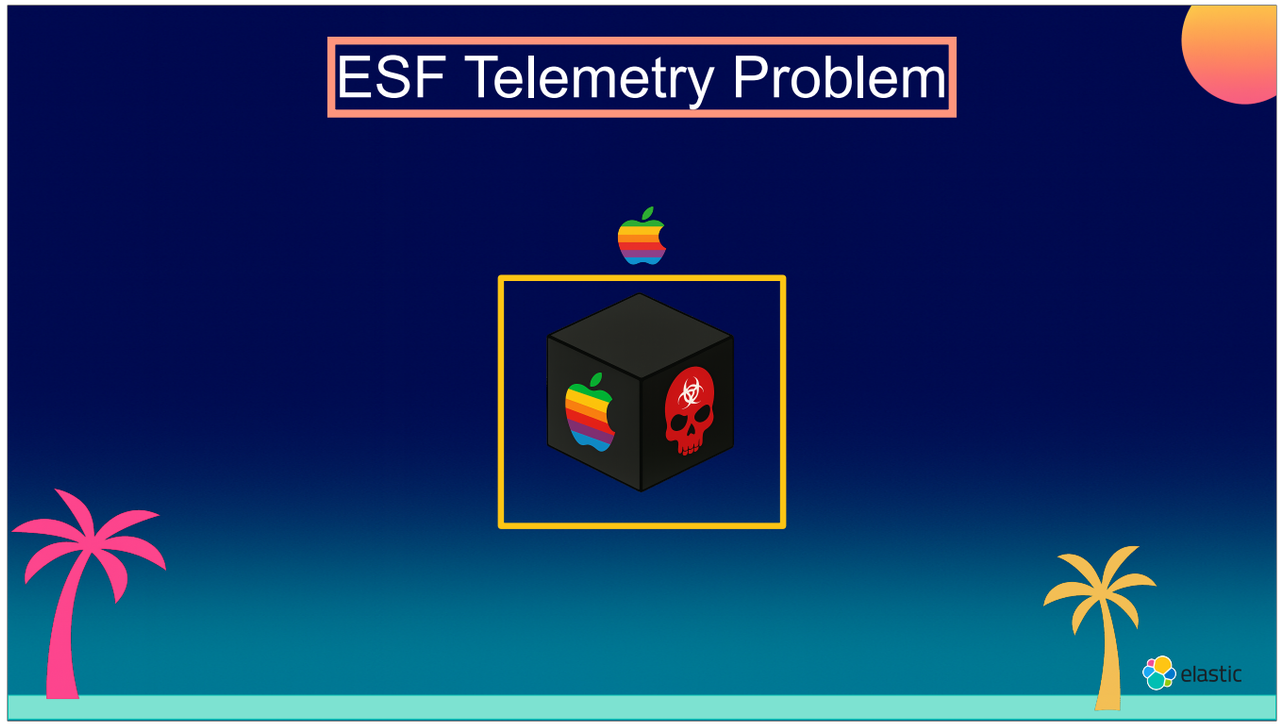
If you collect xpc_connect events you can create a detection rule that looks something like this, inspecting for xpc connections from non-Apple binaries to Container Framework XPC services. This activity should be very rare and definitely warrants an investigation.



In this case we can definitely look for Container process sensitive file access and outbound network connections as this will probably be a lot less noisy than Docker given the Frameworks use will probably be very low and these occurrences rare until its had some time to develop and catch on.

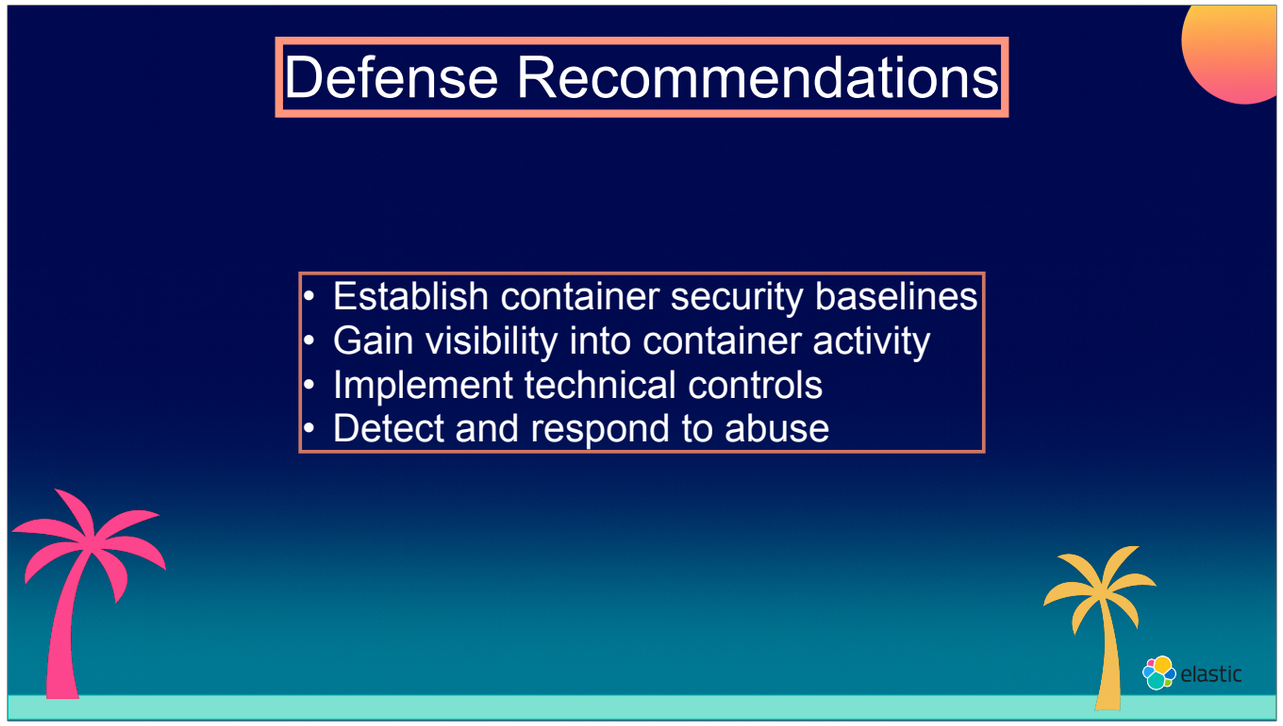


In short, same here, everything the container does is carried out on the host as com.apple.virtualization.virtual.pid a signed, trusted, platform binary. Apple at least includes the PID of the container here but the same problem exists. If we had a container ESF events we could sequence or correlate these actions to container creation but unfortunately we don’t.



So the ways we can detect Container Framework API abuse is very slim and narrow right now though. Surprisingly Apple did not release any new ESF events with MacOS Tahoe and nothing specifically around the Container Framework itself.

I will be submitting a request for container image creation and modification events along with container run and deletion. In the mean time at Elastic we will also be working develop creative solutions to this problem looking to add custom event sources like we have in the past.



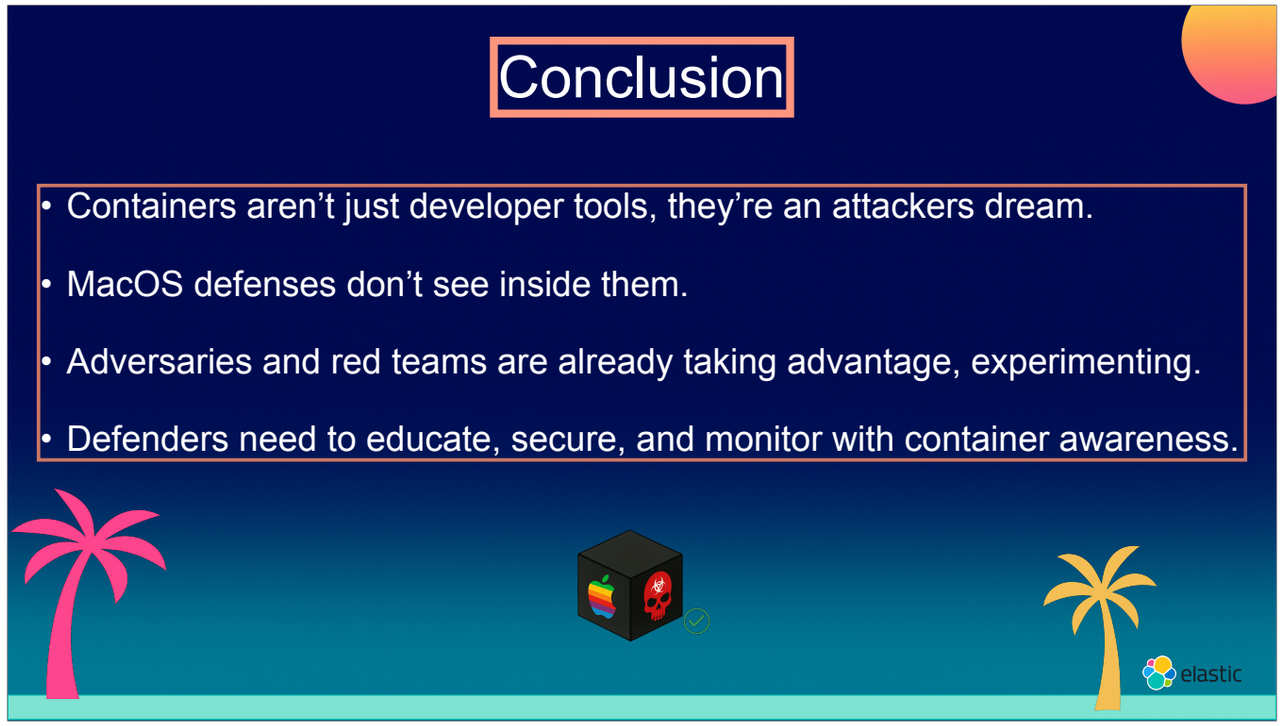
To wrap up the defense section, here are some high level recommendations:

Baseline your companies container security behaviors and activities, easier said than done I know, or at least have some sort of standard way developers should utilize containers so you can observe deviations from that.

Gain visibility into container activity. I mentioned OSQuery and Docker container log collection. Collect those.

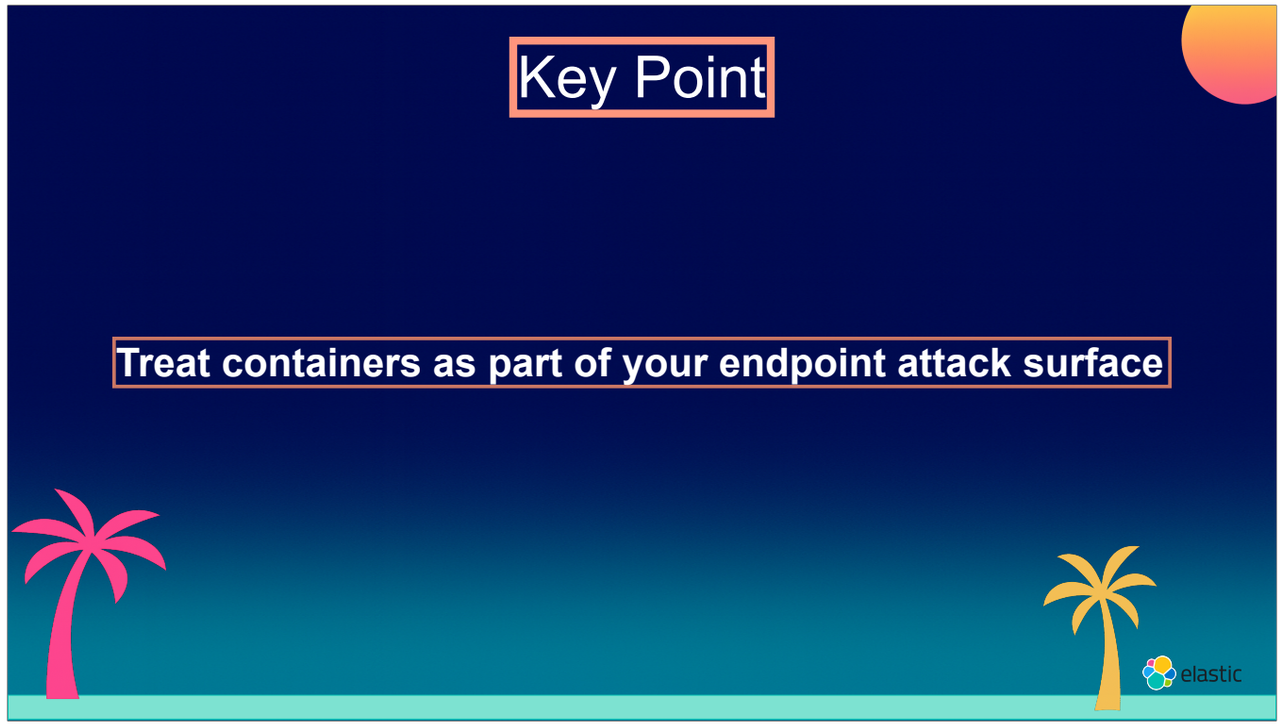
Implement technical controls around your developers container workflows, like Santa for file authorization with Docker settings, restrict permissions, lockdown Docker image retrieval

and finally Deploy detection rules and hunt for anomalies



In conclusion, this is why I called this talk Bring Your Own Blackbox, because once the payload is running inside the container, defenders are blind. Unless we shine a light on it, we’re fighting in the dark.

Containers are a powerful shield for attackers ensuring the many defenses in place on MacOS don’t see their payload. This technique while not having progressed to what I’ve shown you today, at least from what I’ve seen, is still being used successfully and I think we will see more of it in the future especially as the container framework becomes more prevalent.



The key point here is that you need to recognize the danger containers can pose as a tool for adversaries that allows them to quickly and efficiently bypass defenses on MacOS. Start treating containers and the frameworks that run them as part of your endpoint attack surface.

I hope you all enjoyed this talk. This is just a small peak into the research which I hope to publish into a blog or several in the near future so stay tuned. Thank you!