“The Poisoned Diary”: Supply Chain Attacks on Install Scripts

AppSec Village @ DEF CON 29
August 6-8, 2021
@nightwatchcyber
Table of Contents

- Introduction
- Software Supply Chain Attacks
- The “curl | bash” pattern
- Verification of scripts
- Tooling and selected findings
- Takeaways and Q&A
Introduction
In April 2021, it was discovered that a bash script used by “codecov.io” – has been compromised for at least 3 months.

This discovery prompted me to look around: how many other tools that developers use every day are installed that way?

And how can we ensure that these scripts are in fact legit?

I will talk about supply chain attacks but will focus in on downloading and running shell scripts (“curl | bash”)

I also want to talk about how to verify such scripts.

There is also some new tooling and interesting findings that I want to share.
CodeCov.io Compromise

About the Event

Codecov takes the security of its systems and data very seriously and we have implemented numerous safeguards to protect you. On Thursday, April 1, 2021, we learned that someone had gained unauthorized access to our Bash Uploader script and modified it without our permission. The actor gained access because of an error in Codecov’s Docker image creation process that allowed the actor to extract the credential required to modify our Bash Uploader script.

Immediately upon becoming aware of the issue, Codecov secured and remediated the affected script and began investigating any potential impact on users. A third-party forensic firm has been engaged to assist us in this analysis. We have reported this matter to law enforcement and are fully cooperating with their investigation.

Our investigation has determined that beginning January 31, 2021, there were periodic, unauthorized alterations of our Bash Uploader script by a third party, which enabled them to potentially export information stored in our users’ continuous integration (CI) environments. This information was then sent to a third-party server outside of Codecov’s infrastructure.

The Bash Uploader is also used in these related uploaders: Codecov-actions uploader for Github, the Codecov CircleCI Orb, and the Codecov Bitrise Step (together, the “Bash Uploaders”). Therefore, these related uploaders were also impacted by this event.

The altered version of the Bash Uploader script could potentially affect:

- Any credentials, tokens, or keys that our customers were passing through their CI runner that would be accessible when the Bash Uploader script was executed.
- Any services, datastores, and application code that could be accessed with these credentials, tokens, or keys.
- The git remote information (URL of the origin repository) of repositories using the Bash Uploaders to upload coverage to Codecov in CI.
About Me

• I was a software developer most of my career and security bug bounty hunter on the side
• Currently work in application security full time but I’m here personally, not on behalf of my employer
• Presented before at BSides PA/DE/DC & OWASP Global AppSec DC
• Was involved in some early anti-spam work:
  • Co-chaired IRTF’s Anti Spam Research Group (ASRG)
  • Involved in IETF pre-standards work for SPF/DKIM
  • Created protocol for exchanging spam reports (MARF / RFC 5965)
• Helping with the “security.txt” proposal – about to become an RFC
• Also did some non-security standards work:
  • RFCs 4180 (CSV files) and 6922 (SQL MIME type)
  • Participated in W3C’s CSV for the Web group
CVEs I Discovered/Helped With in the Past

- CVE-2021-1000008 (DWF) – Nissan Leaf EV (car)
- CVE-2021-2018 – Oracle Database Server
- CVE-2020-9315 – Oracle iPlanet Web Server (0-day)
- CVE-2020-9314 – Oracle’s iPlanet Web Server (0-day)
- CVE-2019-2114 – Google’s Android OS
- CVE-2019-10320 – Jenkins Credentials Plugin
- CVE-2019-0221 – Apache Tomcat
- CVE-2019-0232 – Apache Tomcat
- CVE-2019-7399 – Amazon’s FireOS
- CVE-2018-15835 – Google’s Android OS
- CVE-2018-9581 – Google’s Android OS
- CVE-2018-9489 – Google’s Android OS
- CVE-2018-6019 – Samsung Display Solutions app
- CVE-2018-0237 – Cisco AMP for Endpoints (MacOS)
- CVE-2017-16905 – DuoLingo’s TinyCards Android app
- CVE-2017-15882 – Private Internet Access Android app
- CVE-2017-15397 – Google’s Chrome OS
- CVE-2017-14582 – Zoho 24x7 Poller for Android
- CVE-2017-13243 – Google’s Android OS
- CVE-2017-11706 – Boozt Android app
- CVE-2017-9977 – AVG AntiVirus for MacOS
- CVE-2017-9245 – Google’s News/Weather Android app
- CVE-2017-9045 – Google’s I/O 2017 Android app
- CVE-2017-8878 – ASUS Routers
- CVE-2017-8877 – ASUS Routers
- CVE-2017-8769 – Facebook’s WhatsApp Android app
- CVE-2017-5892 – ASUS Routers
- CVE-2017-5891 – ASUS Routers
- CVE-2017-5082 – Google’s Chrome browser for Android
- CVE-2016-6936 – Adobe’s AIR SDK and Compiler
- CVE-2016-6723 – Google’s Android OS
- CVE-2016-5672 – Intel’s Crosswalk toolkit
- CVE-2016-5348 – Google’s Android OS
- CVE-2016-5341 – Google’s Android OS
There will be many vendors and OSS projects mentioned in this talk – I am not blaming them – security is hard and they are often working for free

Don’t do anything without talking to a (good) lawyer first!
Software Supply Chain Attacks
Modern software includes many third party OSS and commercial software components
Developer, infrastructure and CI/CD tooling also uses a lot of third party software
Software distribution and updates infrastructure has similar issues (CDNs, web servers, buckets, etc)
Developers and security teams often don’t know what’s inside these components, how they are delivered and if they are legit
Software Supply Chain: Imagination
Software Supply Chain: Reality

From https://xkcd.com/2347/
What if that project is compromised?

Or the maintainer now works for your friendly neighborhood APT?

From https://xkcd.com/2347/
What are Software Supply Chain Attacks?

- A software supply chain attack targets any components or infrastructure used in software development or delivery.
- This can be done by replacing legitimate components or injecting malicious code into them.
- Some attacks target infrastructure instead of the components themselves.
- The goal of the attackers is to gain a foothold inside organizations using these components, to be followed up by something more nasty (ransomware, stealing data, etc).
- Recently, many of these have been done by APTs.
Some examples

115 attacks cataloged in an Atlantic Council report (2020)
Some examples

- Cdnjs security flaw (security research) (2021)
- Dependency confusion (security research) (2021)
- Codecov.io bash uploader (2021)
- SolarWinds Orion (2020)
- Malicious Xcode projects (XCSSET/XcodeSpy) (2020)
- Twilio SDK compromise (2020)
- NotPetya (2017)
- Malicious modules in PYPI / npm / gems / etc (ongoing)
- MageCart attacks (ongoing)

- Many more!!!
Awareness of Your Supply Chain

- Do you know what is in your software? Infrastructure? Tooling? CI/CD? Artifact repository? CDN?
- Do you know where it comes from?
- Who is in charge of the release process?
- This is not an easy problem – often components use other components that use other components
- Same for infrastructure – modern applications will often involve third party CDNs, package stores, clouds, etc.
- For the software you publish and update: do you know who and what’s involved?
Beware of the poisoned diary

- In one of the Harry Potter books, there was a diary that wrote back to you
- Essentially it was running magical “code”
- It turned out to be malicious (surprise!)

“Never trust anything that can think for itself if you can't see where it keeps its brain.”

(Arthur Weasley - Harry Potter and the Chamber of Secrets by JK Rowling)
How to Secure the Supply Chain?

• Consumers:
  • Make sure the component/tool you downloaded matches what the vendor/OSS project produced
  • Make sure what you downloaded is not malicious

• Vendors/OSS projects:
  • Make sure what is being published matches the original source code
  • Make sure what was published is not malicious

• But it is very hard problem!!!
Some approaches to secure the software supply chain

- Outsourcing security to third parties
  - Relying on direct GitHub downloads or source repos
  - Package managers (npm, pypi, etc)

- Cryptography:
  - Checksums
  - OS digital signatures (Gatekeeper, rpm/apt/yum, etc)
  - PGP signatures
  - Public CA ecosystem (Authenticode on Windows, etc)

- Other: A/V scanning, disabling automatic automatic updates, perimeter defenses, sandbox detonation, manual review, etc.

- Experimental:
  - Reproducible builds – really really hard
  - Blockchain
The “curl | bash” pattern
What is this pattern?

- This pattern uses “curl | bash” or “wget | bash” to execute a script downloaded from the Internet in your shell
- It’s a convenient way to install software – especially when bootstrapping larger projects
- There is a long standing debate how safe this is
  - IMHO it’s not since it executes untrusted code in your shell
- Historically, security concerns were downplayed:
  - “Users will review the scripts before running them”
  - “These scripts aren’t used often” (but with CI/CD they are!)
  - “A malicious actor can compromise other artifacts”
“Users will review the scripts before running them” (can you grok shell?)

From XKCD
“Users will review the scripts before running them”

How did Codecov learn of this event?

A customer reported this to us on the morning of April 1, 2021. This customer was using the shasum that is available on our Bash Uploader to confirm the integrity of the uploader fetched from https://codecov.io/bash.

Once the customer saw a discrepancy between the shasum on Github and the shasum calculated from the downloaded Bash Uploader, they reported the issue to us, which prompted our investigation.

- A script used by thousands was compromised and not detected **for three months**!!!
- Detected only because the attacker forgot to change the checksum files
- How many users actually read the bash script???
"Users will review the scripts before running them" (so let’s give them base-85 encoded Python)
“Users will review the scripts before running them” (so let’s replace the shell script with a binary)

Our New Uploader (Beta)

We’ve created an entirely new Uploader using NodeJS that is shipped as a static binary executable on the Windows, Linux, Alpine Linux, and macOS operating systems. Currently, this uploader is in beta, but most standard workflows that are currently accomplished with the Bash Uploader can be accomplished with the new Uploader.

How do I Use the New Uploader?

All releases of the new Uploader are available at https://uploader.codecov.io. Use can be pinned to a specific version, or the latest version can be fetched with the appropriate URL. Despite being in beta, many of the command line arguments present in the Bash Uploader are present in the new Uploader as well. The simplest use case for the uploader is to curl it and use it as follows:

For Linux:

curl -0s https://uploader.codecov.io/latest/linux/codecov
chmod +x codecov
./codecov -t ${CODECOV_TOKEN}
“These scripts aren’t used often” (this code is from Dependabot – a tool for supply chain security)

```bash
206 207  # Install Rust 1.51.0
208 209 ENV RUSTUP_HOME=/opt/rust \
210  CARGO_HOME=/opt/rust \
211  PATH="${PATH}:/opt/rust/bin"
212  RUN mkdir -p "$RUSTUP_HOME" && chown dependabot:dependabot "$RUSTUP_HOME"
213  USER dependabot
214  RUN curl https://sh.rustup.rs -sSf | sh -s -- -y \
215  && rustup toolchain install 1.51.0 && rustup default 1.51.0
216
217  ### Terraform
218
219  USER root
220  ARG TERRAFORM_VERSION=1.0.0
221  RUN curl -fsSL https://apt.releases.hashicorp.com/gpg | apt-key add -
222  RUN apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com ${lsb_release -cs} main" \
223      && apt-get update -y \
224      && apt-get install -y --no-install-recommends terraform=${TERRAFORM_VERSION} \
225      && terraform -help \
226  && rm -rf /var/lib/apt/lists/*
```
“Never trust anything that can think for itself if you can't see where it keeps its brain.”
After the codecov.io breach, I wanted to know how common this pattern is – I started collating a list on GitHub.

I also wanted to find out how to verify these scripts if I wanted to be more secure than an average user.

… So …

This pattern is common.

Some tools provide more secure installation methods such as package managers.

And verification of such scripts isn’t trivial.

Do you know where this pattern is used in your organization???
Some examples

### Projects WITHOUT cryptographic verification

---

**Definition of the fields appearing below:**

- **Name** - name/link to the project
- **Description** - short description
- **Script URL** - URL(s) of the bootstrap scripts
- **Mitigations** - documentation on possible possible mitigations
- **Known issue** - link to a public bug report regarding improving verification of downloads

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Script URL</th>
<th>Mitigations</th>
<th>Known issue?</th>
</tr>
</thead>
<tbody>
<tr>
<td>azure cli</td>
<td>Cloud management tool</td>
<td><a href="https://aka.ms/InstallAzureCli">https://aka.ms/InstallAzureCli</a></td>
<td>Use package managers</td>
<td></td>
</tr>
<tr>
<td>chef</td>
<td>Configuration management tool</td>
<td><a href="https://omnitruck.chef.io/install.sh">https://omnitruck.chef.io/install.sh</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>codecov</td>
<td>Code coverage tool</td>
<td><a href="https://codecov.io/bash">https://codecov.io/bash</a></td>
<td>Checksums</td>
<td>Yes</td>
</tr>
<tr>
<td>dell system update</td>
<td>Firmware update tool</td>
<td><a href="https://linux.dell.com/repo/hardware/dsu/bootstrap.cgi">https://linux.dell.com/repo/hardware/dsu/bootstrap.cgi</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dotnetcore install</td>
<td>Language runtime</td>
<td><a href="https://dot.net/v1/dotnet-install.sh">https://dot.net/v1/dotnet-install.sh</a></td>
<td>Package managers</td>
<td></td>
</tr>
</tbody>
</table>
Some examples – Azure CLI

Install or update

Both installing and updating the CLI requires re-running the install script. Install the CLI by running `curl`.

```
curl -L https://aka.ms/InstallAzureCli | bash
```

The script can also be downloaded and run locally. You may have to restart your shell in order for changes to take effect.
Some examples – Dell firmware upgrade

Dell EMC System Update (DSU)

Starting with DSU 1.7 information on DSU is located at the following link:

- Dell technical resource
- Note: Starting with DSU 1.7, consent is required prior to installing public keys on target systems.

Repository setup

ChangeLog from previous repository: Here

To configure the repository, use the following commands:
1. curl -O https://linux.dell.com/repo/hardware/dsu/bootstrap.cgi
2. bash bootstrap.cgi

Note: Consent is required prior to installing public keys on target systems.
Some examples – Google’s Firebase

```bash
#!/usr/bin/env bash

# This script allows you to install the latest version of the
# "firebase" command by running:
#
# curl -sL firebase.tools | bash
#
# If you do not want to use this script, you can manually
# download the latest "firebase" binary.
#
# curl -Lo ./firebase_bin https://firebase.tools/bin/linux/latest
#
# Alternatively, you can download a specific version.
#
# curl -Lo ./firebase_bin https://firebase.tools/bin/linux/v7.2.2
```
Some examples – pip

Installing with get-pip.py

⚠️ Warning

Be cautious if you are using a Python install that is managed by your operating system or another package manager. get-pip.py does not coordinate with those tools, and may leave your system in an inconsistent state.

To manually install pip, securely ¹ download get-pip.py by following this link: get-pip.py. Alternatively, use curl:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

Then run the following command in the folder where you have downloaded get-pip.py:

<table>
<thead>
<tr>
<th>Unix/macOS</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>python get-pip.py</td>
<td></td>
</tr>
</tbody>
</table>
Some examples – Rust

Installing Rust

You can try Rust online in the Rust Playground without installing anything on your computer.

Rustup: the Rust installer and version management tool

The primary way that folks install Rust is through a tool called Rustup, which is a Rust installer and version management tool.

It looks like you’re running macOS, Linux, or another Unix-like OS. To download Rustup and install Rust, run the following in your terminal, then follow the on-screen instructions. See "Other Installation Methods" if you are on Windows.

```
curl --proto '="https"' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```
Attack vectors

- Scripts like all software travel from the original source repository to some distribution point, often a web server.
- Attackers can target anything along the route including:
  - Webservers / CDNs (cdnjs research)
  - Cloud buckets (Twilio SDK and codecov)
  - Build systems (SolarWinds)
  - Original source code repository (PHP)
  - Credentials of people who have access to any of these
- These are novel attacks since shell scripts aren’t as common as other software components but they have outsize impact.
- Just think what would happen if Rust installer or homebrew are compromised...
Verification of Scripts
What a user wants to do the right thing?

- Now that I know that this pattern is common – how do we verify these scripts are legit?
- Verification, especially with cryptography / PGP is not trivial
- As part of my research, I developed tooling to make this easier and ran it for several months against multiple projects
- I also reached out to multiple projects and commercial vendors, asking them to add cryptographic verification
- Most ignored or refused to implement this request
- My automated tooling brought up some problems as well
- So what can a user do if they need to be secure and still use these scripts?
Don’t use install scripts

• The easiest way to avoid attacks using scripts is not to use them
• Many projects provide more secure alternatives such as OS/language package managers (yum/apt/rpm/npm/pip)
• There are also official releases / binaries available
• While those aren’t perfect, they are harder to compromise than a random web server or S3 bucket (like the Twilio SDK case)
• Formal release management narrows down the pool of people who can produce binaries – scripts are often not managed this way
• But this is inconvenient and isn’t always practical
Going straight to the source

- Another approach is to check out the original repository or download the script directly from the repository.
- Relies on Linus’s law: “given enough eyeballs, all bugs are shallow”
- Since there are many developers and users, someone will see it.
- The flip side is that only one of the committers need to be compromised, makes it easier for an attacker (unlike releases which are controlled by a smaller number of people).
- Changes to source take effect immediately and can start attacking users (unlike releases or versioned packages) – branching helps.
- Some may choose to review the source manually or save a known good version – but most install scripts aren’t versioned.
Going straight to the source - example
Some projects publish checksums – these are often SHA (but MD5 is still in use)

Checksums are useless for security since an attacker can modify them as well BUT it does create another barrier

In the codecov case, the attacker DID NOT modify the checksums and this is how the attack was detected (after 3 months – since most users didn’t verify checksums)

Hosting checksums on a separate server or inside the code repository provides another control (now the attacker needs to compromise two servers)

Same goes for the approach of comparing the script against another server – attacker needs to do more work
Doing it better – cryptography

- Some OSes provide ability to sign scripts with a CA-issued certificate (Authenticode). Relies on existing root CA certificates in the OS – certificate can be stolen or fake ones issued
  - Chocolatey package manager for Windows uses this approach
  - PGP can also be used to generate a signature for the script itself or checksums of the script
- PGP is not CA based, and is notoriously hard to use
- Key distribution is the biggest challenge:
  - Hosting a key on the web means it can be compromised
  - Hosting a key in a PGP key server is vulnerable to key poisoning
  - Haven’t seen attacks involving a compromised key yet
Doing it better – cryptography

```powershell
$schocoPkgDir = Join-Path $schocoPath -ChildPath 'lib\chocolatey'
$snpkg = Join-Path $schocoPkgDir -ChildPath 'chocolatey.nupkg'

if (-not (Test-Path $schocoPkgDir -PathType Container)) {
    $null = New-Item -ItemType Directory -Path $schocoPkgDir
}

Copy-Item -Path $file -Destination $nupkg -Force -ErrorAction SilentlyContinue

#endregion Install Chocolatey
```

# SIG # Begin signature block
MII2vwYJKOZIhvceNAQcCoIIzDCCGawCAQExDzANBgIghkgBZQMEAgEFADB5Bgor
BgEEAYI3AgEEOGswaTA0BgorBgEEAYI3AgEeMCYCAwEAAAQQHw7YFlLCE63JNLG
KX7zUQIBAIABAIBAIBAIBADxMA0GCWQCSAF1AwQCAQUABCBIXT1bVQoJeX
rGtqATyaDxeEHl6Q2pKb3p02Iq/tc6CCFKwggT+MII5d5ADeACEAnANQrvgjQrI
/2BAiAcUAPdMA0GCSqGSIb3DQEBCwUAMHIxCzAJBgNVBAYTA1VTMRUwEwYDVQQK
EwxEaWdpQ2yvCDBjmbMxGTAXBgNVBAETEHd3dy5kaWdpY2YvdC5ja20xMTAvBGNV
BAMTKRERP21LZXJ0IFNITIgQNzdXJlZCBRCBUaW1lc3RhbXBhcmQgQ0EwHhcN
MjEwMTA2MDAwMjQwMzExMTA2MDAwMjQwMjEwMTA2MDAwMjQwMzExMTA2MDAwMjQw
A1UEChMORGlnaUNIcnqzE1uYy4xIDAeBgNVBAAMFROQrZ2lDZXJ0IFRpbWVzdGFt
CcAyMIDxMIIBIjANBgkqhkiG9w0BAQEFAAOCQA8AMIIBCg KsięKEwJUzHgfFivUN
CKRfyMNetuc6EUK9Cn1T2SODFClJhH+HchvkmWsMlucAExjROW/m2HNFZIFwrlj/
Zwucy/02aol6KfjK3CF3fgY38H35x20JpBpqdfopir34hF00dskknx2201PR
0dNaNo/Go+EvGczqYdz7E5M4mp8UUtS7FQ5kE61naG3MjJfQd0jeWkst3Tjyj9X
Tycg6W0LUNuj2VRNeEbjA4MxKUpcDDGKSoiyXfcsWvkmUxvVfbENJCF0m11P2jJW0
GqtbSR0wwptgrTb/FZuVb+hh6u+elsKIC9L4cmVp4y+y+Zj0i61chzun3oBc/gZ
1v4NSYS9AIDAQAB0s4B1DCBepqBdqgYrDVD0PAQH/BAQDAgeAAMAwGA1UdEwEB/wQC
MAAwFyzYDYR0LAQH/BAawCgYIKwYBQUHAgwQQYDVR0gBDowODA2BgglhkAgBhv1s
BwEwKTTAnBcDrBoEFBoCARYbHR0cDoVl3d3dv5kaWd0Y2VdC5ib20v01BTM8B
```
The quickest way to install Nix is to open a terminal and run the following command (as a user other than root with sudo permission):

```
$ curl -L https://nixos.org/nix/install | sh
```

Make sure to follow the instructions output by the script.

The installation script requires that you have sudo access to root.

You may want to verify the integrity of the installation script using GPG:

```
$ curl -o install-nix-2.3.14 https://releases.nixos.org/nix/nix-2.3.14/install
$ curl -o install-nix-2.3.14.asc https://releases.nixos.org/nix/nix-2.3.14/install
$ gpg2 --recv-keys B541D55301270E0BCF15CA5D8170B4726D7198DE
$ gpg2 --verify ./install-nix-2.3.14.asc
$ sh ./install-nix-2.3.14
```

The signing key has fingerprint B541 D553 0127 0E0B CF15 CA5D 8170 B472 6D71 98DE. It is also available on GitHub.
Summary of approaches

- Avoiding shell scripts and using package managers or releases might be the safest option?
- Checksums, downloading from source or doing a comparison against another version helps somewhat as long as the attacker doesn’t get access to both places
- Cryptography works best but is hard to setup and use
- Key distribution for PGP is not easy, CA certificates have their own issues as well
- Most projects I surveyed don’t provide signatures
- Organizations need to make their own decisions based on risk and security posture
- More work / research is needed!
Tooling and Selected Findings
The state of tooling

• After the codecov.io breach, I wanted to now how common this pattern is and what are safer alternatives
• I also wanted to see how easy it would be to verify scripts in an automated fashion, suitable for a CI/CD
• Another problem was to see what happens when scripts get updated – is there a lag?
• Last, I wanted to monitor scripts for changes to detect similar attacks in the future
• The existing tooling was sort of lacking … so I wrote up some documentation and created my own
The state of tooling

- The best way to do this using tools like GPG directly
- But they are not easy to use
- GPG also changes machine state (keyrings) which may not always be desirable
- They are also not easily configurable and I wanted a way to provide configuration files for many projects I am monitoring
My experimental tools

- Documentation I created:
  - **dont_curl_and_bash** - list of projects using the curl/bash pattern, plus different ways to verify downloads
  - Plus some blog posts and this talk!

- Tools I created:
  - **icetrust** - tool for verification of downloads
  - **icetrust_dashboard_example** - monitoring example using icetrust
  - **icetrust_uptime_example** - “status page” example of monitoring using icetrust
  - **release_auditor** - tool for checking if GitHub releases were modified after initial publication (see this blog post)
Icetrust tools

- “icetrust” is a Python module that lets you verify files you already downloaded by comparing them against something else, using a checksum or via PGP signatures.
- It also has a “canary” mode which will download the files for you, is controlled using a JSON config file and can produce a JSON output file, all intended for automation.
- The two dashboard examples are using GitHub pages / Jekyll and the “upptime” project to periodically poll scripts and check for changes. Results are also piped to Twitter.
- Monitoring uses GitHub actions under the hood and I keep copies of the files retrieved so the changes can be reviewed over time via standard Git tools.
Example: don't curl and bash

## Projects WITHOUT cryptographic verification

Definition of the fields appearing below:

- **Name** - name/link to the project
- **Description** - short description
- **Script URL** - URL(s) of the bootstrap scripts
- **Mitigations** - documentation on possible possible mitigations
- **Known issue** - link to a public bug report regarding improving verification of downloads

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Script URL</th>
<th>Mitigations</th>
<th>Known issue?</th>
</tr>
</thead>
<tbody>
<tr>
<td>azure cli</td>
<td>Cloud management tool</td>
<td><a href="https://aka.ms/InstallAzureCli">https://aka.ms/InstallAzureCli</a></td>
<td>Use package managers</td>
<td></td>
</tr>
<tr>
<td>chef</td>
<td>Configuration management tool</td>
<td><a href="https://omnitrack.chef.io/install.sh">https://omnitrack.chef.io/install.sh</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>codecov</td>
<td>Code coverage tool</td>
<td><a href="https://codecov.io/bash">https://codecov.io/bash</a></td>
<td></td>
<td>Checksums</td>
</tr>
<tr>
<td>dell system update</td>
<td>Firmware update tool</td>
<td><a href="https://linux.dell.com/repo/hardware/dsu/bootstrap.cgi">https://linux.dell.com/repo/hardware/dsu/bootstrap.cgi</a></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>dotnetcore install script</td>
<td>Language runtime</td>
<td><a href="https://dot.net/v1/dotnet-install.sh">https://dot.net/v1/dotnet-install.sh</a></td>
<td>Package managers</td>
<td></td>
</tr>
</tbody>
</table>
Example: icetrust – using regular mode

```
pgp

First download the software to be verified and its signature file:

curl -O https://www.example.com/software.zip
curl -O https://www.example.com/software.zip.sig

Verify using a key ID:

icetrust pgp software.zip software.zip.sig --keyid 12345 --keyserver pgp.example.com

If you want to use a keyfile, you must download it or provide it, then verify:

curl -O https://keys.example.com/project_keys.txt
icetrust pgp software.zip software.zip.sig --keyfile project_keys.txt
```
Example: icetrust – canary mode

To use this mode, you must create a JSON file that follows the defined schema then pass it to the tool via command line:

```
icetrust canary config.json
```

If you want to output the details to a JSON file, do the following:

```
icetrust canary --output-json-file output.json config.json
```
### Verification results - using checksums or PGP

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Verified?</th>
<th>Recently Changed?</th>
<th>Additional Details</th>
<th>Last Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>codecov_uploader</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:37 +0000</td>
</tr>
<tr>
<td>nixos_installer</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:47 +0000</td>
</tr>
<tr>
<td>rvm_installer</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:52 +0000</td>
</tr>
<tr>
<td>salt_bootstrap</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:53 +0000</td>
</tr>
<tr>
<td>volta</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:56 +0000</td>
</tr>
</tbody>
</table>

### Verification results - comparing against another file

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Verified?</th>
<th>Recently Changed?</th>
<th>Additional Details</th>
<th>Last Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>azure_cli</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:36 +0000</td>
</tr>
<tr>
<td>heroku_cli</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:41 +0000</td>
</tr>
<tr>
<td>ibm_cloud_cli</td>
<td>✓</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:42 +0000</td>
</tr>
<tr>
<td>jfrog_cli</td>
<td>✗</td>
<td></td>
<td>config / output / file</td>
<td>2021-07-23 18:20:43 +0000</td>
</tr>
</tbody>
</table>
Example: icetrust_dashboard – Tracking file changes

```bash
#!/usr/bin/env bash

# Apache License Version 2.0, January 2004
# https://github.com/codecov/codecov-bash/blob/master/LICENSE

set -e +o pipefail

VERSION="1.0.6"

codecov_flags=

url="https://codecov.io"

env="$CODECOV_ENV"
```
## Example: icetrust_uptime

<table>
<thead>
<tr>
<th>Service</th>
<th>Overall Uptime</th>
<th>Average Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>azure cli</td>
<td>100.00%</td>
<td>129 ms</td>
</tr>
<tr>
<td>codecov uploader</td>
<td>99.55%</td>
<td>45 ms</td>
</tr>
<tr>
<td>heroku cli</td>
<td>100.00%</td>
<td>38 ms</td>
</tr>
<tr>
<td>ibm cloud cli</td>
<td>100.00%</td>
<td>36 ms</td>
</tr>
</tbody>
</table>

### Open and Closed Issues
- **codecov uploader is down**: #83 by nightwatchcyber was closed 2 days ago
- **codecov uploader is down**: #82 by nightwatchcyber was closed 8 days ago
- **codecov uploader is down**: #81 by nightwatchcyber was closed 11 days ago
- **nixos install script is down**: #80 by nightwatchcyber was closed 18 days ago
Example: Twitter account

Icetrust
@icetrustmonitor
Icetrust - detect supply chain attacks
🔗...dashboard.nightwatchcybersecurity.com
Joined May 2021
0 Following    0 Followers

Tweets

Icetrust @icetrustmonitor · Jul 21
homebrew: file content has changed (2021-07-21 18:21:03 +0000)

Icetrust @icetrustmonitor · Jul 21
codecov: file content has changed (2021-07-21 18:21:03 +0000)

Icetrust @icetrustmonitor · Jul 21
codecov: verification failed (2021-07-21 18:21:03 +0000)

Icetrust @icetrustmonitor · Jul 20
sdkman: file content has changed (2021-07-20 18:21:02 +0000)
How I used these tools

• I documented over 25 projects using pattern
• There are dozens more I reached out to via security disclosure processes that I can’t speak publicly about yet
• Monitoring signatures on 5 projects
• Comparing against another copy – 11 projects
• Simply monitoring for changes – 14 projects
• Monitoring runs every 6 hours (used to be 1 hour)
• Results are piped to the two dashboards, a Twitter account ( @icetrustmonitor) and email/Slack
• Looking into incorporating icetrust into my own projects as a safeguard when scripts are used
• Has been running since April/May
What did I find?

- Only a minority of projects offers a way to verify their scripts cryptographically (via PGP) – with the key hosted elsewhere
- Most projects/vendors are not interested in a fix
- Many projects offer better alternatives but it’s not consistent
- The use of “raw” GitHub downloads is common
- Most bootstrap scripts do not change often, or change in minor ways only (version numbers, etc)
- Verification of some is not trivial (requires custom parsing)
- Sometimes there is a lag between a new script being available and its signature being recalculated/updated – automation fails:(
- Often there is no formal release management or versioning
- No supply chain attacks found … yet

**BUT … it is possible to monitor on an automated basis!**
Some incidental findings

- A bug in GitHub that allows releases to be modified without anyone knowing about it (unless you use APIs). This is why I wrote release_auditor. BTW - GitHub rejected the bug.

- Google’s Firebase CLI calling Google Analytics when installed

- Some scripts do not match their source code repository exactly (in minor ways)

- Research is still ongoing!
Takeaways and Q&A
Takeaways

- We need to have more awareness about what is in our software, tooling and build processes (remember the poisoned diary!)
- The use of bootstrap scripts is dangerous, safer alternatives may exist but everyone needs to make their own risk decisions
- It is possible to verify some scripts but its not always trivial – you must be prepared for things to break
- If your organization publishes scripts or any kind of software, consider monitoring them for unauthorized changes
- Formal release management is important – less people have access, formal approval, two men/four eyes rule, etc.
- Use native tooling – mine is experimental (but fun!)
- Don’t forget the rest of the software supply chain – this is just one tiny piece!
Questions? Comments?

Email: research@nightwatchcybersecurity.com