Secure Programming Techniques

Margus Freudenthal, Mart Oruaas

Lecture 4, 2022-03-11
Software Integrity, Supply Chain security
Two-way Street

- Supply chain security is a two-way street
- How can I trust the software I use and/or redistribute?
- How can I make verification of the software I create and distribute the easiest for my users?
Third Party Components
Third Party Components

- Nobody is writing software from scratch
- We make extensive use of third-party software
  - Operating Systems
  - Middleware (databases, app engines, web servers)
  - Frameworks (Spring, React)
  - Libraries
- **Most of the code in your system is not written by you**
- That code can contain vulnerabilities
- That code can contain **intentional** vulnerabilities, which are very hard to unveil [Thompson, 1984]
Open vs. Closed Source Security

- The intuitive thinking is that open source software can be more secure, because of the "many eyeballs" advantage.
- Alas, research shows no significant difference between open and closed source security [Ozment and Schechter, 2006]. The difference comes down to how big and capable is the team, how seriously they take security, and how actively the component is maintained.
- I recommend you to take a small project of about 10 thousand lines of code and try to review it so that you can make some solid claims about its security. Try to understand how many hours of work that is.
- OpenSSL Heartbleed bug was undiscovered for 3 years.
Spot the Heartbleed!

```c
hbtype = *p++;
n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version,
                 &s->s3->rrec.data[0], s->s3->rrec.length,
                         s, s->msg_callback_arg);

if (hbtype == TLS1_HB_REQUEST)
{
    unsigned char *buffer, *bp;
    int r;

    /* Allocate memory for the response, size is 1 bytes
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    bp += payload;
    /* Random padding */
    RAND_pseudo_bytes(bp, padding);

    r = ss13_write_bytes(s, TLS1_RT_HB, buffer, 3 + payload + padding);
```
Evaluating Third Party Components

- Don’t just add a new dependency to your project
- You need to evaluate the new component to assess future costs and risks
- There is an inherent tension:
  - It is bad to let your application’s build chain to update all dependencies on its own (see UAParser.js incident from October 2021),
  - Yet you must be ready to update components on a short notice (see Log4j incident from December 2021)
One possible checklist for evaluating third party components has been published by Software Sustainability Institute in the UK [Jackson et al., 2011]. It is derived from product quality criteria listed in ISO 25010:2011 [ISO, 2010]. Lists **eighteen** criteria for making a decision about a software component – vetting software is not easy.
Some of the criteria have direct impact on the component’s security posture:

- Understandability: easily understood?
- Documentation: comprehensive, appropriate, well-structured user documentation?
- Learnability: easy to learn how to use its functions?
- Testability: easy to test correctness of source code?
- Analysability: easy to understand at the source level?
Updating Components 1/2

- One must keep track of the external dependencies used in the product
- Software Bill of Materials (SBOM) – maybe not the best idea, but a step in the right direction
- In the most critical cases, tracking may go down to operating system and CPU microcode level
- Be ready to update your software fast
- Be ready inform your customers (in-house or external), they are the ones that take the actual risks
Updating Components 2/2

- Maintain high quality buildchain
- Maintain in-house binary artefact repository, use only vetted components from there (both external and internal)
- Use Software Composition Analysis (SCA) tools:
  - OWASP Dependency Check (Java, .NET)
  - retire.js (JavaScript)
  - npm-audit (JavaScript)
- GitHub has built-in security scanning tools, at least on some paid service levels
Supply Chain Towards Customers
Securing the Software Delivery 1/2

- Manual signing, binary digest verification etc works only in very specialized cases
- Luckily, all modern widespread computing platforms have supply chain protection mechanisms baked in, with verification going down to driver, kernel and bootloader level
  - iOS: App Store
  - macOS: code signing and notarisation
  - Android: Play Store, albeit can be easily bypassed
  - Windows: code signing on all levels
  - Linux: signatures in package managers
Unfortunately, this is where the good parts end

- No modern web application delivery system has built-in or at least widely used verification mechanism
- Those systems can be huge and very much unobservable
- The setups very often violate Zero Trust principles
- Namespaces for software packages are global, which open up possibilities for unexpected code injections [Birsan, 2021]
Advanced Techniques

- Reproducible builds
- Static and dynamic analysis of source code and compiled artefacts
- It most likely requires commercial outsourcing, but companies like SynopSys (Coverity) and Veracode know what they are doing
- Veracode’s “State of Software Security” is a very interesting read [Veracode, 2022]
Supply Chain Attacks
Supply Chain Attacks

- The chain is as strong as its weakest link
- With all the third party components used, the attack surface is massive
- Lots of libraries of primitives are not guarded all that well and some programming languages have systemic problems [Zimmermann et al., 2019]
- People have been aware of the problem for quite some time but have chosen to ignore it [Thompson, 1984, Chess et al., 2007]
- Ohm et al researched a number of more recent incidents in 2020 [Ohm et al., 2020]
XKCD Has a Pic for Every Situation

The infamous *left-pad* npm package in 2016.
Attack Vector Examples

- Typosquatting – hoping that the target makes a typo in dependencies’ declaration
- Brandjacking/trojan horse – create a new package with reasonable name (twilio-npm vs twilio)
- Dependency confusion [Birsan, 2021]
- Use after free – taking up a discontinued project/package/user
- Sending a pull request – adding new functionality and an external dependency (controlled by the attacker)
- Compromising the repository – stealing credentials, exploiting vulnerabilities, setting up fake mirrors, etc.
Zero Trust
Zero Trust

“Zero Trust” is a bit of a misnomer, $\epsilon$-trust would be a more appropriate name.

- It bears similarities with both Dolev-Yao security model and Kerckhoffs’s principle.
- The main idea is in the contrast with older, perimeter-based security models, where being inside the perimeter meant full access and almost limitless privilege of repudiation – the area inside the perimeter is fully trusted.
- M&M Security – “a hard crunchy shell (if any) with a soft chewy center”
Zero Trust in the US Federal Government

- US federal government published a set of recommendations about Zero Trust in January 2022 [OMB, 2022]
- This may have an impact on the whole scene because US government footprint is huge
- Specifically tells that Zero Trust is **not** giving up on perimeter security, but assuming that it does not exist
Zero Trust has direct impact on the software development

- Identity management must be properly implemented
- Strong phishing-resistant authentication must be implemented
  - There are only two options: ID-card-like solutions and WebAuthn
- Strong authorization architecture must be in place
- Pervasive encryption, which in turn means proper key management
- Scopes of protection domains are reduced, pushing security functions to applications
Questions?
References I

https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

References II


References IV

Moving the U.S. Government Toward Zero Trust Cybersecurity Principles.

Milk or Wine: Does Software Security Improve with Age?
References V
