Understand how integrating external components changes your attack surface

It is unlikely that you will develop a new system without using external pieces of software. In fact, when adding functionality to an existing system, developers often make use of existing components to provide some or all of that new functionality. In this context, external components refer to software "not written here", such as:

- Software procured as off-the-shelf components, platforms, and applications
- Third-party open source or proprietary libraries
- Widgets and gadgets added or loaded at runtime as part of a web project
- Access to some functionality provided by the component that you plan to take advantage of (e.g., accessing a web service that provides federated authentication)
- Software developed by a different team within your organization
- Software that your team developed at a previous point in time (perhaps at a time when the security stance was not as mature as it is now)

These components may be included as binaries, libraries, and source code, or they may exist simply as APIs.

It is a common adage of software security that whenever possible, functionality should be achieved by the reuse of tried-and-true pieces of previously tested and validated software, instead of developing from scratch every time. The important distinction is that the software being newly included has actually been tried as well as tested and found to stand up to your current standards of software security. The decision to use-rather-than-build means that the software as a whole inherits the security weaknesses, security limitations, maintenance responsibility, and the threat model of whatever you are including. This inheritance can amount to a deficit of security, which must be solved, mitigated, or accounted for when the system is finished. The system's "threat model" is a representation of the security posture of the system when all possible threats are taken into consideration, their mitigations established, and the vulnerabilities identified.

Make sure you allocate time in your software development methodology to consider the security impact on your system when including an external component:

- How does the external component change the threat model of the entire system? Does it add to the attack surface? Does it modify entry points in the system that had already been considered in its own threat model?
- Were new features, capabilities, or interfaces added even though you are not using them? Can those unused features be disabled?
- Does the external component being included also include other external components with their own security weaknesses?
- Have you obtained the external component from a known, trusted source?
- Does the external component provide security documentation that may help you better understand its threat model and the security implications of its configuration?

You must assume that incoming external components are not to be trusted until appropriate security controls have been applied, in order to align the component's attack surface and security policy with ones that meet your requirements.

Examples of potential security issues with third-party components include the following:

- Loading a library with known vulnerabilities (CWE, CVE, etc.)
- Including a library with extra features that entail security risks
- Reusing a library — yours or a third party's — that no longer meets current software security standards
- Using a third-party service and hoping thereby to pass responsibility of security to that service
- Configuration mistakes in the security of a library — e.g, secure defaults
- Library making outbound requests to the maker's site or to some partner of theirs
- Library receiving inbound requests from some external source
- A single external component including other components, causing multiple levels of inclusion ("recursion")
- Including pieces of functionality that offer unknown interfaces into the system – for example, a CLI for configuration of an included daemon, a panel or admin mode for a Web component, a hardcoded set of credentials for an authentication/authorization module, a debugging interface or backdoor, or the like.

At a minimum, consider the following:

- Isolate external components as much as your required functionality permits; use containers, sandboxes, and drop privileges before entering uncontrolled code.
- When possible, configure external components to enable only the functionality you intend to use.
- If you include functionality that you do not intend to use, you must consider how that included functionality changes your security posture (attack surface, inherited debt, threats, etc.), and therefore increases the security you must implement to account for the change.
- If you cannot configure the security properties of the component to align with your security goals, find another library, or document that you are accepting the risk and inform relevant stakeholders of your decision.
- Likewise, if the element to be included cannot realize your security objectives, find a different element, or document that you are accepting the risk and inform relevant stakeholders of your decision.
- Validate the provenance and integrity of the external component by means of cryptographically trusted hashes and signatures, code signing artifacts, and verification of the downloaded source. If no integrity mechanism is available, consider maintaining a local mirror of the library's source. Understand the risk of dynamically including components such as JavaScript from external sources. If the external host is compromised you may be including attacker-controlled JavaScript.
- Identify and follow sources that track or publish security-related information regarding the external components you consume: bug repositories, security-focused mailing lists, CVE databases, and so forth.
- Make sure that the development team members charged with responding to security events are aware of all external components used so those can be included in their threat intelligence collection efforts.
- Maintain an up-to-date list of consumed external components and at a pre-established cadence verify that it matches the versions included in your product, as well as that those are the latest known-secure versions available for each external component.
- Maintain a healthy distrust of external components:
  - Whenever possible, authenticate the dataflow between your system and external components.
  - Consider all data coming from an external component to be tainted, until proven valid (see "*Define an approach that ensures all data are explicitly validated*" for additional information).
- Be sure to understand and verify the default configuration of the external component. For example, if you are including an external crypto library, understand what values are used by default unless you change them; for example, sources of entropy, algorithms, and key lengths. When consuming an external component such as a Web server, understand its defaults concerning admin modes, ports where the processes will be listening, and assumptions concerning how it interfaces with the operating system and with

your own software.

- Document everything. If you change a default, make sure that there is documentation as to why the decision was made to change it. If you include an external component, create documentation around the process used to choose the component, the provenance of the component, the verification it went through, and most importantly any security-relevant assumption made about it. This will make it easier to move forward when versions change, or when you consider the use of an alternative external component. When changing the build defaults of external components, configuration options for deployment, or source code, automate the procedure using your version control system or a patch file (numerous tools, including make, sed, and patch, are available for this task depending on your environment). Then include the automated procedure in your build workflow - bring in the pristine component, apply your modifications, and use it for your build. The automation will help to maintain consistency between builds, and some tools include calling modes or executables that validate their own configurations; leverage those into your process as well to know when your modifications need adjustment due to a version change in the external component or some other similar event.
- Design for flexibility. Sometimes an external component becomes too risky, or its development is abandoned, or the functionality it offers is surpassed by another external component. For those cases, you will want to design your system so that external components can be easily replaced.