1. Criteria for critical software

David A. Wheeler <dwheeler@linuxfoundation.org> (et al.)
Director of Open Source Supply Chain Security, The Linux Foundation (*willing to speak*)

> **NIST Request:** *Criteria for designating "critical software."* Functional criteria should include, but not be limited to, level of privilege or access required to function, integration, dependencies, direct access to networking and computing resources, performance of a function critical to trust, and potential for harm if compromised. See EO Section 4(g).

Critical software could be defined as software that presents an especially high level of risk should it fail or be compromised, where risk is a combination of impact and likelihood:

1. *Impact*. Many issues control the impact of software.
    a. *Safety-critical*. If the software fails or is compromised it could lead to death or serious injury to people; exposure of privacy protected information; loss of financial assets; identity theft; national security issues; compromised infrastructure; loss or severe damage to equipment/property; and/or environmental harm. While this in many ways is the "best" measure of impact, past experience shows that doing *only* this analysis directly is unlikely to correctly determine impact, due to many challenges:
        i. Software is typically part of larger systems. The exact composition of software in larger systems is often unknown (the "top" may be known but not its transitively decomposed parts).
        ii. Systems also typically depend on other tools (including build tools) and services, which again may depend on other tools and services transitively, and all those services also typically have a large set of transitively-included software and hardware components.
        iii. Systems may have countermeasures that attempt to reduce the impact of problems, but it's often unclear how well those countermeasures work in practice against intelligent adversaries.

    Thus, indirect measures (as noted below) are often important to help identify software that is high-impact but might otherwise be missed.
    b. *Category of software* (type of function or data). Functions that are keys to trust are especially important. These include cryptographic libraries (which, if subverted, can quietly subvert an entire system and typically have cryptographic keys) and anything with elevated privileges such as operating system kernels. Software in such categories only matters if they're used. However, it's often difficult to determine *where* they are used if they're used at all.
    c. *Ubiquity*. If the software is widely used, its subversion can lead to widespread damage and make repair especially difficult. As an analogy: A billion gallons of water can be more dangerous than one. There have been various ways to estimate this. One approach is to examine dependencies to find the software that is "most depended on by others" (e.g., from a sample set of programs analyzed or across entire ecosystems). Tracking "GitHub stars" measures software project popularity of a kind but is dubious as a measure of ubiquity. More promising than stars is tracking

"watchers" of OSS; watching something takes time. Since attention is limited, that is a better measure than stars for software that is likely to be important (or becoming important).

2. *Likelihood*. The likelihood that software could be subverted includes factors such as:
    a. *Network accessibility & attack surface*. It is often easier to attack software if it is directly exposed to a network or directly processes untrusted data. If the software exposes many attackable interfaces, again, it's easier to attack.
    b. *Development process*. If the development process aggressively looks for vulnerabilities (using tools and people) and proactively fixes them and/or designs to prevent them, the likelihood of easily exploited vulnerabilities is reduced.
    c. *Memory safety*. Software that in non-memory-safe languages have several additional attack risks that memory-safe languages typically do not.
    d. *Size*. Programs with a smaller codebase are much easier to analyze.
    e. *Physical security*. Many attacks take advantage of access to data, software and hardware. This can include lack of a full trusted execution environment or even signal leakage.
    f. *Review*. Software that others have not reviewed is riskier. One of the potential advantages of OSS is that it enables mass peer review.

Different organizations, governments, and societies will have a different set of critical software.

Here are some related projects and/or information sources:

1. "Open Source Software Projects Needing Security Investments" aka "Census I" by the Institute for Defense Analyses (IDA) and the Linux Foundation analyzed Linux distribution packages (specifically Debian) to identify critical software.
2. Linux Foundation and the Laboratory for Innovation Science at Harvard (LISH) developed the report Vulnerabilities in the Core,' a Preliminary Report and Census II of Open Source Software, which analyzed the use of OSS to help identify critical software. The LF and LISH are in the process of updating that report.
3. The LF Core Infrastructure Initiative (CII) identified many important projects and assisted them, including OpenSSL (after Heartbleed), OpenSSH, GnuPG, Frama-C, and the OWASP Zed Attack Proxy (ZAP).
4. The OpenSSF Securing Critical Projects Working Group has been working to better identify critical OSS projects and focus resources on critical OSS projects that need help. There is already a proposed list of such projects, along with efforts to discuss funding such aid. Although they are nascent, you may find the following materials helpful: Table-Top Exercise: What is Critical Software? along with Table-Top Exercise: List of Categories of Critical Software.