Threat Model

Automated software vulnerability discovery, exploitation, and patching

Henry Post, hp2376@nyu.edu, NYU-CS-GY-6813

The threat, in this problem domain (Automated software vulnerability discovery, exploitation, and patching) comes from external systems that would automatically scan open-source software (OSS) source code, parse it, detect vulnerabilities, and then attempt to exploit those vulnerabilities. It could also include human attackers that perform the same function as the external systems – exploiting software bugs.

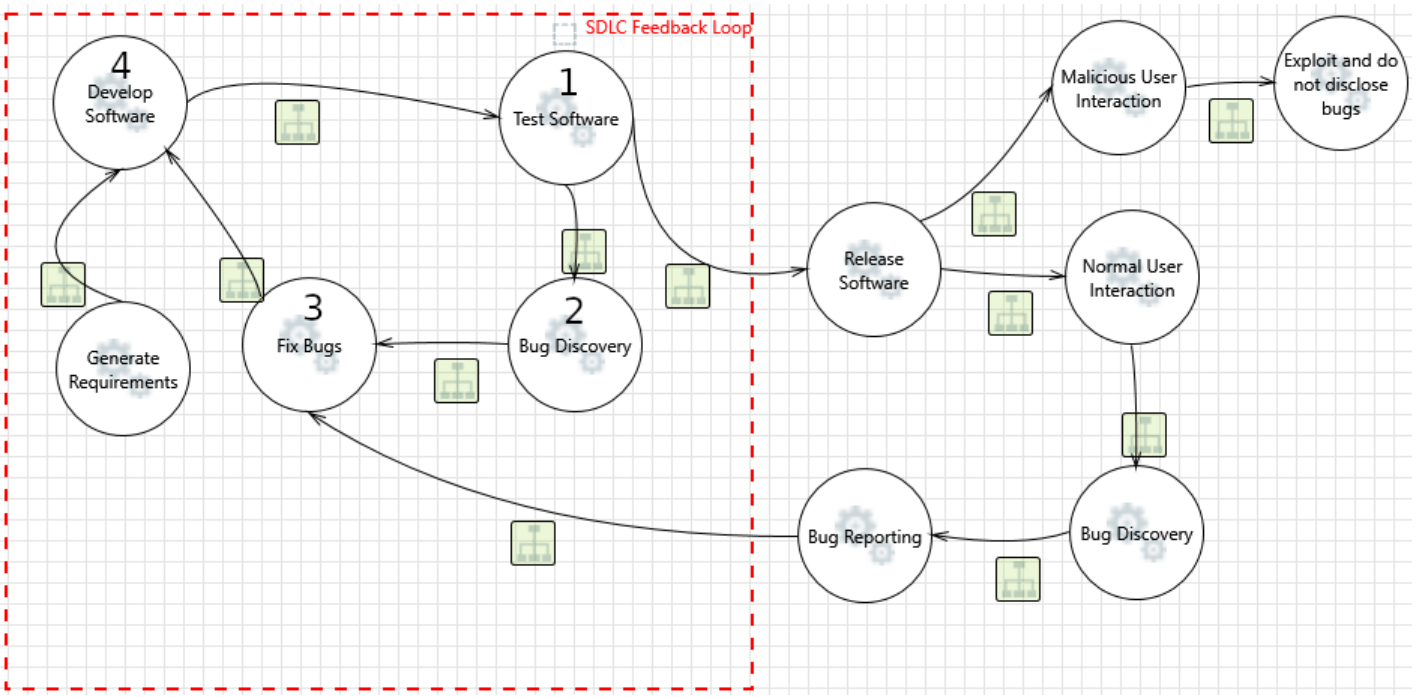
We need to protect the customers that consume the OSS itself from undiscovered bugs in the software. The vulnerability exists because vulnerable software is released at all, in any capacity. Attackers take advantage of the time that it takes maintainers of OSS to detect vulnerabilities, patch the code, and then deploy patches to customers. This time is valuable to an attacker, as it means that they can attempt to exploit the vulnerability in systems that are unpatched.

“Customer” in this context means any individual, organization, or even other software components that consume the OSS. For example, a “customer” could be you, if you used Apache Tomcat (webserver software) to host a personal website. It could also be Canonical Ltd., the current maintainers of the Ubuntu distribution of Linux, because they likely use nginx (network load balancer software) to host various websites, and nginx is OSS.

Ideally, the vulnerabilities would be detected earlier in the software development lifecycle (SDLC), before code was ever released to customers. This can be done by performing automated security scanning earlier, and generally improving the security posture of the entire SDLC (“shifting security left”).

Attackers have various methods and motives for targeting OSS, and the attack methods are very diverse. They occur after we release the vulnerable software into the wild. The OWASP Top Ten list is a good starting point to understand what kind of attacks could be performed on insecure software.

The diagram below shows a very simplified view of the SDLC lifecycle, focusing on vulnerability generation and how vulnerabilities are exploited and fixed. Note that in this diagram I use “bug” and “vulnerability” interchangeably.



Our goal is to prevent vulnerabilities from being introduced in releases ("Release Software"). In order to do this, we need to ensure that vulnerabilities are always discovered **within** the SDLC Feedback Loop (red box) and not outside of it.

Everything outside of the red box (past "Release Software") will have an effect on customers. If we allow vulnerabilities outside of software releases, they could end up negatively affecting the people that use software that we develop.

We cannot rely on our users to report on vulnerabilities, and we cannot rely on malicious users to not exploit vulnerabilities. Therefore, we must ensure that we do not introduce vulnerabilities outside of the SDLC.

The fact that bugs ever leave this "simplified" SDLC Feedback Loop could be viewed as a failure of one or more of the processes within the loop. To fix this, one or more nodes (processes) must be modified, or one or more nodes must be added/removed.

1. Develop Software
    - This is where bugs are generated.
    - The fact that bugs are generated here is a failure of this node.
        - Improve developer training/security mindset.
        - Replace human developers with automated (AI/robot) developers that are less error-prone.
        - Improve development processes
            - Scan code as it is run in a test system

            - Scan code while it is written inside the IDE (editor) by a developer
            - Scan code as it is compiled
            - Utilize defensive programming/paradigms
                - Design-by-contract, Fail-fast, Strong typing, Typestate analysis, Formal verification, etc.
2. Test Software
    - This is where bugs **should** be detected and fixed.
    - The fact that bugs are not detected here is a failure of this node.
        - Improve testing procedures
            - Unit testing
                - Require full code coverage
        - Improve software analysis
            - Static application security testing, (SAST)/Dynamic AST/Integrated (dual) AST scanning
            - Scan code after it is checked-in to a version control system (VCS) and before it is released
            - Scan third-party components that are used in code
                - Detect CVE (Common Vulnerability Enumerations) that have been reported for third-party components
3. Bug Discovery
    - This is the process of realizing a bug exists, recording its existence, and/or fixing it.
4. Fix Bugs
    - This is the process of performing code changes to fix a vulnerability (bug).
    - This process feeds back into (1), “Develop Software”.

In summary, the entire SDLC lifecycle could be improved with additions to software testing, packaging, code validation, code scanning, analysis of third-party components used in source code, and improved developer training on security and secure coding. Many points made in this Threat Model can be summed up by “Move security sooner into the development process”, or “shift security left”.