Attacking the Build through Cross-Build Injection

How Your Build Process Can Open the Gates to a Trojan Horse

Brian Chess, Fredrick DeQuan Lee, Jacob West
{bchess,flee,jwest}@fortify.com
September 10, 2007

Summary

A poorly designed software build process can allow an attacker to insert malicious code into the final product or to take control of a build machine. This paper surveys previous attacks related to building open source software, including attacks against Sendmail, OpenSSH and IRSSI. It then shows how three popular build tools for Java (Apache Ant,[1] Maven[2] and Ivy[3]) are commonly misused in ways that make them susceptible to cross-build injection (XBI) vulnerabilities, which can allow attackers to insert Trojans, back doors, or other malicious code.

1. Introduction

The most damaging attacks on software are the ones that allow the attacker to execute code on the compromised system. For a security researcher, the phrase "allows remote code execution" is akin to saying "checkmate". For example, buffer overflow attacks are so fearsome because they often allow the attacker to take control of the target program by inserting new code while the program is running. In this paper we investigate cross-build injection vulnerabilities (XBI)—vulnerabilities that allow attackers to insert code into the target program while the program is being constructed.

No one builds software from scratch anymore. As Figure 1 illustrates, modern software projects are almost always built from a combination of internally written source code and externally developed components. In many enterprise environments, it is now normal for at least some of these external components to be open source.

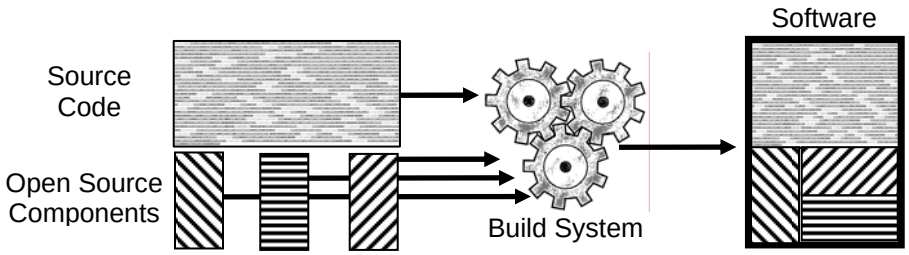

Figure 1: Software built from source code and open source components

External dependencies in general, and open source components in particular, do not necessarily represent an unacceptable security risk, but external dependencies do deserve

[1] http://ant.apache.org/
[2] http://maven.apache.org/
[3] http://incubator.apache.org/ivy/

vetting in order to ensure that they are non-malicious and acceptable for the purpose at hand.

An automated and repeatable system for compiling code is a hallmark of a mature software development process. Automating the build process reduces the likelihood of human error and increases the maintainability of the project. However, the benefits of an automated build process can also lead to systemic problems. If the build process automatically retrieves external dependencies such as open source components, then an attacker has an opportunity to insert code into the program by compromising the external dependency. Figure 2 illustrates how the traditional build process in Figure 1 can be altered by a build system that performs automated dependency management.

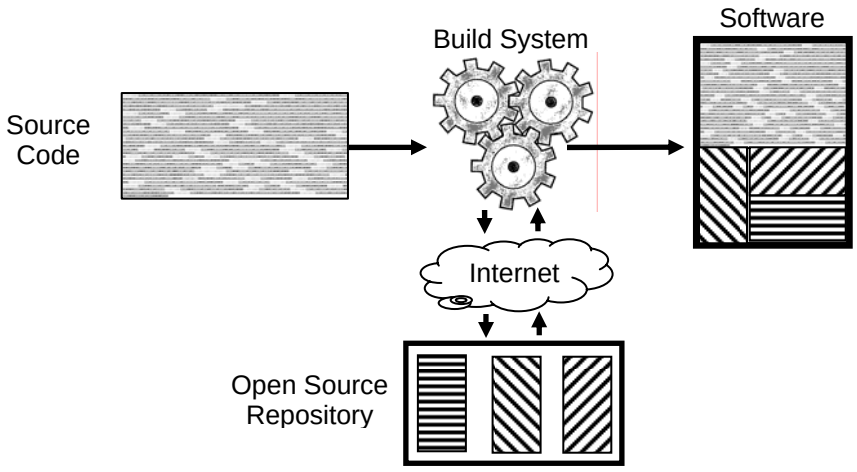

Figure 2: Software built from source code and open source components

Section 2 provides the background for this work by reviewing a number of security incidents where attackers have subverted the build process. Section 3 provides an overview of the external dependency features of three Java build tools: Ant, Maven and Ivy. Section 4 shows how these features are often misused and discusses the ways attackers can turn this misuse to their advantage. Section 5 provides recommendations for using the build tools discussed in Section 3 in a secure fashion, and Section 6 concludes the paper.

2. Open Source Build Security Incidents

A number of incidents involving compromised open source dependencies have come to light in the past several years. These compromises had several similarities: they occurred in open source projects; they involved an attacker subverting the canonical source repository for the project and inserting malicious code; and the compromises affected users that traditionally installed the components through an automated build system.

In this section we detail compromises in OpenSSH, Sendmail and IRSSI. These incidents demonstrate that attackers have already identified the build process as a way to compromise systems. The attacks abuse users' trust in the authenticity of packages hosted on external servers and their adoption of automated build systems that encourage this practice. These attacks were successful because of poorly founded assumptions, the most

significant of which was the idea that the security of an external repository is sufficient to meet the security needs of the end user.

OpenSSH 3.4p1 Trojan

OpenSSH is a component that plays a fundamental role in the security of many systems. CERT advisory CA-2002-24[4] details the introduction of malicious code into versions 3.2.2p1, 3.4p1, and 3.4 of OpenSSH. On July 31st, 2002 a hacker compromised servers for ftp.openssh.com and ftp.openbsd.org and inserted malicious code designed to provide backdoor access to an attacker.

Compromised versions of OpenSSH spread to users who did not download them directly thanks to the OpenBSD component installation tool, *port*. The port system was developed to offer a consistent, repeatable, and automated mechanism for building and installing OpenBSD components. To install software on OpenBSD, a user issues a system command that specifies a package name. The port system then retrieves the package from the OpenBSD server and compiles it automatically.

In this case, the source retrieved from the external entity was malicious and included a modified Makefile that linked in attacker-controlled source code from the file bf-test.c. Once the Makefile was executed, the malicious code was compiled and run as part of the install process, which allowed an attacker to connect to the compromised machine with the same permissions as the user compiling the code. Since the normal procedure on the OpenBSD systems was to install software as a privileged user, the attacker was given privileged remote access.

Sendmail 8.12.6 Trojan

For many years, Sendmail was the de facto mail server on the Internet. From September 28, 2002 until October 6, 2002 a compromised server at ftp.sendmail.org was distributing a modified source distribution of Sendmail version 8.12.6 that contained a Trojan horse. The malicious software had been put in place by an attacker who infiltrated the Sendmail server and altered the build process for the source distribution for Sendmail to, during the build process, open a socket to host 66.37.138.99 on port 6667.

CERT advisory CA-2002-28[5] described the impact of this exploit as follows:

> "An intruder operating from the remote address specified in the malicious code can gain unauthorized remote access to any host that compiled a version of Sendmail from this Trojan horse version of the source code. The level of access would be that of the user who compiled the source code."

Any user who built the compromised source distribution of Sendmail was open to attack. The compromised system created a tunnel to the remote system of the attacker, which

[4] http://www.cert.org/advisories/CA-2002-24.html
[5] http://www.cert.org/advisories/CA-2002-28.html

gave the attacker unauthorized remote control of the victim's machine. If root or another privileged user compiled Sendmail, the attacker gained complete control of the compromised host.

IRSSI 0.8.4 Backdoor

On May 25, 2002 a message posted to the BugTraq mailing list[6] explained that a Trojan horse had been embedded in IRSSI, an open source IRC client. An IRSSI user discovered that IRSSI version 0.8.4 included the following malicious backdoor code:

```
int s;
struct sockaddr_in sa;
Forks the backdoor code as a new process
switch(fork()) { case 0: break; default: exit(0); }
if((s = socket(AF_INET, SOCK_STREAM, 0)) == (-1)) {
  exit(1);
}
/* HP/UX 9 (%#!) writes to sscanf strings */
memset(&sa, 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(6667);
sa.sin_addr.s_addr = inet_addr("204.120.36.206");
Opens a network connection to the remote address 204.120.36.206
if(connect(s,(struct sockaddr *)&sa, sizeof(sa)) == (-1)) {
  exit(1);
}
dup2(s, 0); dup2(s, 1); dup2(s, 2);
```

The code above forks and opens a socket to a remote server at 204.120.36.206 (presumably controlled by the attacker). The new socket gives remote access to systems running the vulnerable version of IRSSI, including users who installed the compromised version of the software via the source package, such as Fink users on Mac OS X.

3. Handling External Dependencies with Maven, Ivy, and Ant

The examples discussed in the last section validate the fact that external component repositories can and will be compromised by attackers and, in the case of the OpenSSH and IRSSI exploits, users who build external components using an automated system have an even greater exposure. We now turn our attention to how modern build systems handle projects that rely on external dependencies and consider the security ramifications of these systems.

Unmanaged Dependencies Make Projects Hard to Build

In a manual build process, the person building the code is typically required to provide any third party components the software being built requires. Under such a model, the

---

[6] http://archives.neohapsis.com/archives/bugtraq/2002-05/0222.html

software being built has very little control over how its external dependencies are resolved. The external dependencies might be available from multiple origins, and any one dependency might be available in different versions or from multiple distribution sources.

When it comes to replicating a build environment across multiple environments, unmanaged dependencies can cause trouble. A build engineer might download a component from the wrong source, download the wrong version of a component, or forget to download a necessary component entirely. Any of these issues can cause the build to fail. To overcome this risk, manual builds often include explicit directions on the specific version of each component and the repository from which it should be retrieved

For example, consider a hypothetical piece of software and two engineers, Dick and Jane, who must build the software independently. The process for completing a manual build might resemble the following:

1. Dick determines the necessary dependencies for the project and updates the build file with the appropriate references.
2. Dick then downloads the required dependencies from external servers, stores them on a local build machine and points the build system at the files.
3. Dick builds the software on his local build machine.
4. Jane attempts to deduce the external dependencies from the build file.
5. Jane downloads what she thinks are the correct dependencies from external servers, stores them on a different local build machine, and attempts to build software. Jane repeats this step until the build succeeds.

Automatically Resolving External Dependencies

Several tools exist within the Java development world to aid in dependency management: both the Ant and Maven build systems include functionality designed to help manage dependencies, and the primary purpose of Ivy is to serve as a dependency manager. Although there are differences in their behavior, these tools all allow a developer to specify that external dependencies be resolved by automatically retrieving code at build time and compiling it into the software being built.

The advantage of automatic dependency resolution is that it minimizes the effort involved in building a project for the first time and ensures that the project is built using the same components each time. Developers need only store dependency information in the build file, compile the code, and deploy, without the dependency management hassles involved in a manual build. The following examples give specific details for how Ant, Maven and Ivy allow developers to specify external dependencies.

Ant: Developers specify external dependencies in an Ant target using a `<get>` task, which retrieves the dependency from the corresponding URL. This approach is functionally equivalent to the scenario where a developer documents each external dependency as an artifact included in the software project, but the `<get>` task is desirable because it automates the retrieval and incorporation of dependencies when a

build is performed. The following excerpt from an Ant build.xml file shows a typical reference to an external dependency:

```
<get src="http://people.apache.org/org/apache/commons-
logging/commons-logging-1.1.jar"
dest="${build.repo.local}/org/apache/commons-
logging/commons-logging-1.1.jar"
usetimestamp="true" ignoreerrors="true"/>
```

Maven: Instead of listing an explicit URL from which to retrieve dependencies, developers specify dependency names and versions and Maven relies on its underlying configuration to identify the server(s) from which to retrieve them. For commonly used components this saves the developer from having to research dependency locations. The following except from a Maven pom.xml file shows how a developer can specify an external dependency using its name and version:

```
<dependencies>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

Maven allows the component provider to specify a MD5 hash for each component, which is also retrieved from the component repository at build time. This provides a way for Maven to verify the integrity of the software, but it does not vouch for the origin of the software. In other words, it protects against any sort of file corruption that might occur during download, but it does not protect against an attacker who has access to the repository. An attacker who can replace a real component with a malicious one can just as easily provide a valid MD5 hash of the malicious component.

Ivy: In addition to the functionality provided by Maven, Ivy adds the flexibility to allow the developer to forgo specifying a version of the dependency and instead to list a version status, such as latest released version. The following excerpt from an Ivy ivy.xml file shows a dependency that relies on a version status:

```
<dependencies>
  <dependency org="apache"
              name="commons-logging"
               rev="latest.released" conf="default->*"/>
</dependencies>
```

Ivy supports essentially the same MD5 capabilities as Maven.

4. Attacks against Ant, Maven and Ivy

The wide-spread adoption of modern build systems that include dependency management capabilities has made attacks similar to the ones discussed in Section 2 increasingly

powerful and easy to carry out. Two distinct types of attack scenarios affect these systems: An attacker can either compromise the server that hosts the component or compromise the DNS server the build machine uses to redirect requests to a machine controlled by the attacker. Both scenarios can enable an attacker to inject a malicious version of the dependency into a build running on an uncompromised build machine.

Figure 3 illustrates how an attacker might first take over an external component repository and then use that compromise to take over a machine that uses a component.

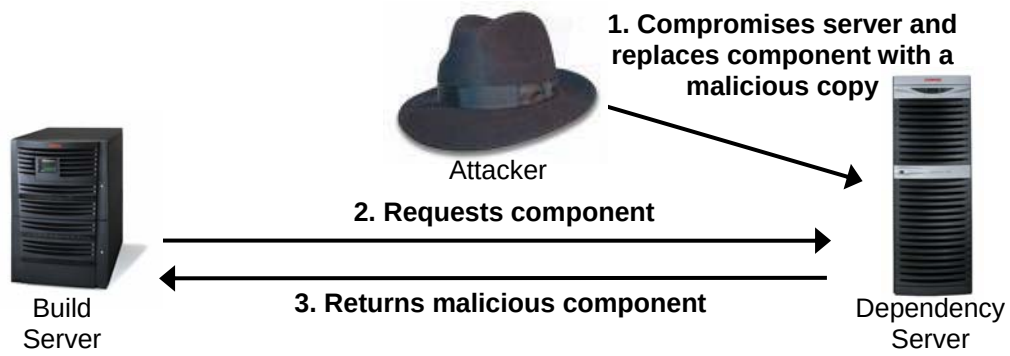

Figure 3: Attacker replaces legitimate dependency with a Trojan

Figure 4 shows how an attacker could inject a malicious component by compromising the DNS record the build machine uses to locate the dependency server.

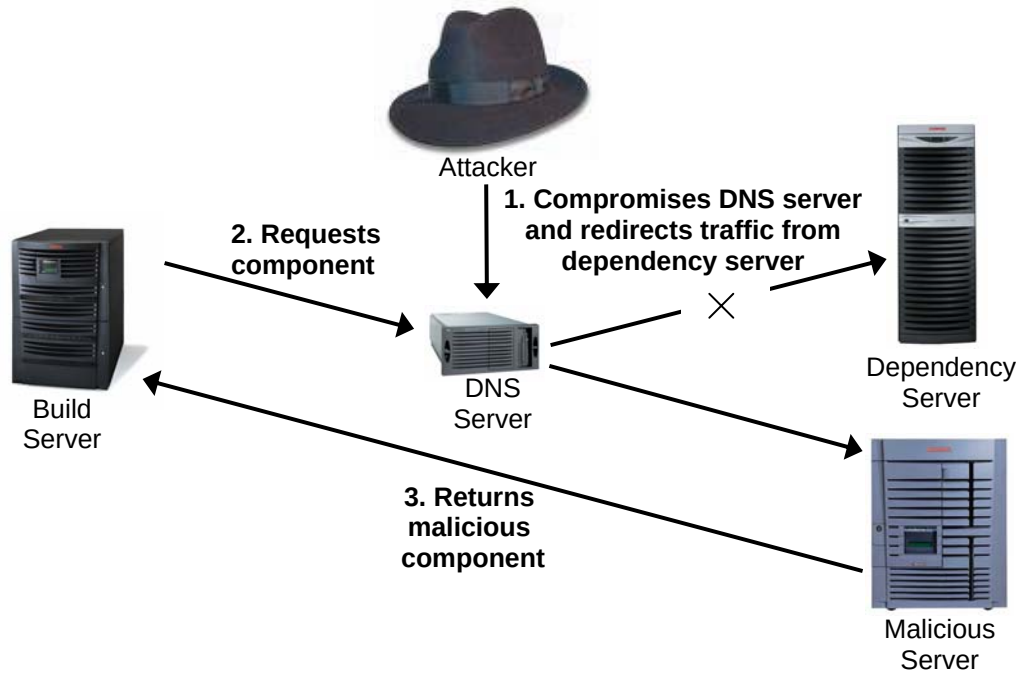

Figure 4: Attacker redirects DNS requests to a malicious server

Regardless of the attack vector used to deliver the Trojan dependency, both scenarios share the common element that the build system blindly accepts the malicious component and includes it in the build. Because the build system does not have the capability to

detect the malicious component, and existing security mechanisms, such as code review, typically focus on internally-developed code rather than external dependencies, this type of attack is likely to go unnoticed as it spreads through a development environment and potentially into production.

There is a significant security risk in any scenario in which someone downloads code of unknown provenance and then executes it without vetting it, but automatic dependency resolution systems make the problem much worse. If a build system retrieves components from external sources each time the build system is run in a new environment, it greatly increases the window of opportunity for an attacker. An attacker need only compromise the dependency server or the DNS server during one of the many times the dependency is retrieved in order to compromise the build machine. This difference in the window of vulnerability between manual and automated build systems is illustrated in Figure 5.

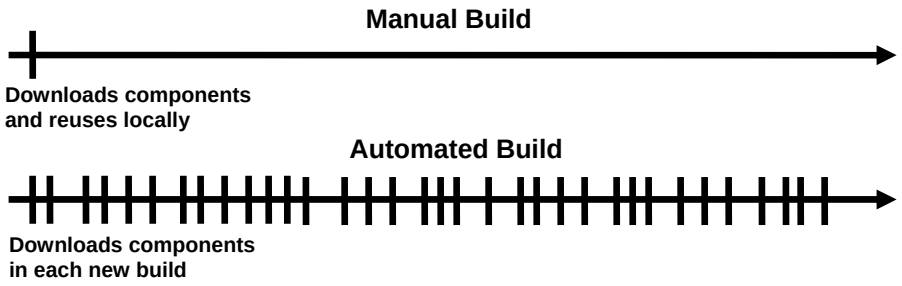

Figure 5: Window of vulnerability in a manual build versus an automated build

5. Recommendations

In this section we propose three approaches to mitigating the risk introduced by automated dependency management systems. The most restrictive solution is to ban the use of these systems altogether. A second, more pragmatic approach is to create a dependency server. The final solution expands upon the second but adds a human review process for dependencies before they are added to the internal repository to reduce the risk of introducing malicious code to the trusted environment. Regardless of the solution you choose, you should include the configuration files that control the build process as part of your code review process.

The simplest solution is to refrain from adopting automated dependency management systems altogether. Managing dependencies manually eliminates the potential for build system to introduce unexpected risks. An attacker could still mount one of the attacks described above to coincide with the manual retrieval of a dependency, but limiting the frequency with which the dependency must be retrieved significantly reduces the window of vulnerability. On the downside, this solution forces the development organization to rely on what is ostensibly an antiquated build system and could decreases the likelihood that open source components will be updated as bugs are resolved and new versions are released. A system based on manual dependency management is often more difficult to use and maintain, and we note that teaching abstinence has been conspicuously ineffective in other contexts.

The second solution is a hybrid of the traditional manual dependency management approach and fully automated solutions, and strikes a good balance between cost and security. Create an internal server to host dependencies. As new dependencies are added to software projects and new component versions are released, download them once and add them to the repository. This solution reduces the window of vulnerability and allows the organization to leverage existing internal network security infrastructure. This is the approach recommended by the Ivy team, who suggest that all enterprises create such an internal server.[7] We note that the DNS server for this internal dependency machine now plays a security-critical role. DNS can be factored out of the equation by referring to the server's IP address rather than its hostname.

To implement this approach in Ant, simply replace the target address for each `<get>` task with the a reference to the internal repository. With Maven, hardcode a reference to the internal repository in the pom.xml for each project. Specifying the name in pom.xml ensures the internal repository will be used by the corresponding build, but is tied to a specific project. Alternatively, the reference can be specified in settings.xml, which makes the configuration easier to share across multiple projects. The following Maven pom.xml demonstrates the use of an explicit internal IP address (the same entries can also be used in settings.xml):

```
<repositories>
  <repository>
    ...
    <id>central</id>
    <name>Internal Repository</name>
    <url>http://172.16.1.13/maven2</url>
    <layout>default</layout>
  </repository>
</repositories>
```

Ivy allows users to specify which servers to retrieve dependencies from through a separate Ivy configuration file (typically ivyconf.xml). The following excerpt from an ivyconf.xml file demonstrates the use of a properly configured internal repository:

```
<ivyconf>
  <properties file="ivyconf.properties"/>
  <conf defaultResolver="default" checkUpToDate="true"/>
  <resolvers>
    <chain name="default">
      <url name="localrepo" checkmodified="true">
        <ivy
pattern="http://172.16.1.13/[org]/[mod]/ivy-[rev].xml"/>
        <artifact
pattern="http://172.16.1.13/[org]/[mod]/[art]-[rev].[type]"/>
```

[7] http://incubator.apache.org/ivy/history/trunk/bestpractices.html

```
      </url>
    </chain>
  </resolvers>
</ivyconf>
```

An Ivy configuration should also specify a version for each dependency, rather than relying on the version status. The simple solution is to force developers to always specify the exact dependency version to be used, as shown in the example above.

The second solution can be further improved to include a security vetting process for external dependencies before they are included on an internal dependency repository. For organizations that already perform security reviews of software, this process might entail performing code reviews for security or other security testing activities on new open source dependencies before they are certified for use within the organization. By preventing build systems from accessing external dependencies from any source other than the internal repository, the organization can ensure that projects do not bypass this security vetting process.

The risk posed by XBI vulnerabilities could be greatly diminished if build tools allowed users to specify a cryptographic signature for each package they use. Checking against a signature would verify that the code originated from a trusted source. This approach is standard fare for patch management. Microsoft Windows and Mac OS X can both be configured to retrieve updates automatically and both operating systems cryptographically verify the origin of the patches they download.

6. Conclusions

Not all software security vulnerabilities appear in source code. As we have seen, attackers know how to use an open source repository to distribute a Trojan horse or other malicious code. The build process is open to attack.

We train all computer users not to execute untrusted programs that arrive via email, but programmers routinely download code from the Internet and build it into the software they create. This risky practice becomes doubly bad when it is automated using a build tool such as Ant, Maven, or Ivy. When a build process is set up to automatically retrieve code from the Internet, the security of the program being built will depend forevermore on the security of the sites hosting the open source components and the network infrastructure used to locate them. Instead of being exposed to potentially malicious code once first and only time a component is retrieved, the build process is exposed every time it returns to the component repository.

The easiest way to avoid XBI vulnerabilities is to ban the use of automatic dependency resolution. If such an austere solution is not feasible, establish an internal repository and a vetting process for bringing new packages into the repository.