Integrating Automated Tools Into a Secure Software Development Process

Kenneth R. van Wyk
KRvW Associates, LLC
Ken@KRvW.com

This paper is intended to augment and accompany its corresponding slide presentation.

Description

Automated security tools are often used in software development, from static source code analysis tools to penetration testing tools. Unfortunately, due to a variety of reasons, many development organizations fail to get the maximum benefit from the tools. Worse, the way that many organizations use security tools may actually hamper effective development work. Penetration testing tools, for example, are commonly used for late life cycle "black box" testing. This forces, at best, knee jerk reactions to remediate any defects that are found, quite often at the expense of the application's original design concepts. It also likely fails to find a great many security defects. To make matters worse, forced integration of tool technologies into existing workflows can be disruptive and counter productive.

This paper delves into the automated tools associated with secure software development, and how they can be successfully integrated into a development workflow.

Tool categories are first surveyed, and their utility and applicability to secure development reviewed. These include traditional information security tools such as network vulnerability scanners and application vulnerability scanners, as well as more focused development-only tools such as static source code analyzers. The pros and cons of each tool set is described in plain detail, with particular attention to how software developers can benefit from them.

Next, individual tool categories are discussed with regards to how they can be integrated into a secure software development workflow process. This portion of the session starts by examining the pitfalls associated with how the tools are often put to use by software developers, and then provides a clear set of recommendations of how to best make use of the tools.

Penetration testing tools (and processes), for example, are often used in a late life cycle approach that "verifies" an application's security level shortly before its deployment into production. This approach is inherently a "black box" one in which the application is assessed in an outside --> in perspective. This talk recommends an alternate approach to using penetration testing tools in an inside --> out manner that optimizes employee time and effort by prioritizing work based on identified business risks. That is, "white box" penetration testing can focus on the aspects of an application that have been identified as being weak during architectural risk analyses.

Similarly, static source code analysis tools are often used in a late life cycle manner that leaves little time for remediation of identified coding defects. In this paper and corresponding session, we explore methods of integrating static source code analysis tools throughout the coding process in a way that greatly optimizes their likelihood of success and reduces the amount of effort necessary.

Overview of software security process "touchpoints"

Information security practitioners have applied security tools in a myriad of ways over the years. However, it's not so frequently the case that software developers have applied security tools to assist them in developing secure software. This is a shame, as there are numerous ways that security tools can be applied throughout the development process to positive effect.

Before we get into the tools themselves, let's start with a very quick overview of common software security practices, often referred to as security "touchpoints".

- Requirements. Most (but not all) software developers collect and document their projects' functional requirements, quite often in the form of "use cases". To look for security shortcomings in the functional aspects of software, security practitioners often turn to "abuse cases" that describe how an attacker could misuse the very functionality of the software itself. Unfortunately, this is still largely a manual process. Although the actual findings should be prioritized and tracked, few tools have proven themselves to be more useful—or popular—for this purpose than a good old spreadsheet.

- Design. Increasingly, software developers are using one or more techniques to evaluate the security of their designs. Although the processes are maturing and improving, few tools are available to help automate this step (beyond the spreadsheet, as in the above case).

- Coding. Code reviews, both manual as well as automated, are the state of the practice for finding coding mistakes, including security ones. Common coding mistakes such as buffer overruns, SQL injection, and cross-site scripting are relatively easy to spot at a source code level. As such, several commercial static analysis tools for evaluating the security of many popular programming languages are now readily available.

- Testing. Security testing is no longer just about penetration testing. Indeed, such outside→in, or "black box," testing should not make up more than just a small percentage of a rigorous security testing regimen. Today's state of the practice security testing includes risk-driven testing of all aspects of the software, from the perspective of having full knowledge of the code, design, etc.

  Automation abounds here, from the traditional IT Security focused penetration testing tools through far more rigorous and in-depth software testing tools and frameworks such as fuzzing.

Survey of existing tools

Similarly, we'll provide very quick overview of the available tools that are commonly used in software development and testing. First, though, let's consider their backgrounds a bit, as this is relevant to how we'll go about using them for development purposes.

Security tools can be placed into two very broad categories: those primarily written for information security practitioners, and those primarily written for software developers. Note that there can be overlap between these, as we continue to see tools that attempt to suit multiple audiences.

Many security tools have been written over the years for information security practitioners. Although quite useful, they often fall short of speaking to the issues faced by software developers. Network vulnerability scanners (discussed in a bit more detail below) are classic examples of this phenomenon. If your job is to assess and control the configuration of the servers on a network, knowing which systems contain unpatched network daemons is entirely useful. If, on the other hand, your job is to secure the software that makes up an application, then the reports generated by network vulnerability scanners are utterly useless, as they contain no actionable recommendations that are relevant to the software developer.

With that in mind, let's start with a short list of tools commonly used by information security practitioners.

- Network security scanners.
- Vulnerability scanners.
- Application vulnerability scanners.

(For an excellent list of tools that fall into these categories, see Fyodor's "top 100" list at http://sectools.org)

Many of these tools have been the mainstays of information security for a decade or so, in varying degree. They automate the process of probing the hosts on a network for poorly configured and maintained network services. Indeed, unpatched vulnerabilities virtually jump off the report page when even the most rudimentary network scanner pores through a network.

The most recent of them, so-called application vulnerability scanners, take the first step at addressing application-specific security weaknesses. They do so by probing an application's (or web service's) external interfaces for common coding weaknesses such as SQL injection. These tests are useful and meaningful, but are fundamentally "black box" tests of an application.

The most significant shortcoming, from the perspective of software development, of black box testing is that it is impossible to ensure even adequate test coverage. (Coverage is a commonly used measure of what percentage of an application is being exercised by its testing. It includes the percentage of actual lines of code executed, error conditions encountered, interfaces exercised, etc.) The most effective way of striving for thorough test coverage is to plan the testing from an inside→out perspective, with full access to source code, designs, etc.

With that in mind, here is a short list of security tools commonly used by software developers.

- Static source code analysis tools.
- Fuzzing tools.
- Dynamic validation and other testing tools.

Static analysis of source code has, as we've said, seen a significant number of commercial tools in the past few years. Indeed, today's commercial static analysis suite is an enterprise-ready tool kit that can greatly assist in pinpointing a myriad of common coding flaws in most programming languages. However, despite the advances in even the best of the commercial tools, a significant amount of work remains for the software developer in analyzing and assessing the tool's findings.

Software "fuzzing" has been rapidly growing in popularity in the past couple years. It is the practice of bombarding a piece (or component) of software with input and observing failure states – which often represent logic failures in boundary condition processing, for example. Numerous fuzzing tools and frameworks are available today that can test a range of fuzzing targets such as file formats, network interfaces, and user interfaces.

The third category of tools and techniques commonly used by software developers is dynamic validation tools. These can include a myriad of different practices such as interposition and such to observe and validate that a piece of software is actually executing the security aspects of its design faithfully.

Integration into development workflow

With so many tools available, one might think that it is trivial to acquire a tool chest and dive in, and get useful and meaningful results. Unfortunately, the state of the practice could hardly be further from this idealistic scenario. Many mistakes are commonly made, and otherwise useful, powerful tools, are often relegated to nothing more than "shelfware". This is truly a shame, as many opportunities for improving things are being missed.

In order to help the reader prevent this "shelfware" experience, let's explore the various processes and how tools can best be integrated into them with a minimum of difficulty.

Penetration testing

The traditional black box approach to penetration testing is fundamentally limited in what it can accomplish for software developers. There are a couple of key issues in getting value from penetration testing in the software development lifecycle[1]. These include the following list:

- Adopt a white box approach.
- Use business risk prioritization to allocate testing time and effort.

To be valuable in the software development process, penetration testing must follow an informed testing process in which the test team has unfettered access to the software's design documentation, architectural risk analysis results, as well as source code. From these documents, reasonable test scenarios can be developed that seek to exercise the weaknesses discovered and theorized during the risk analysis process, for example.

Once the set of test scenarios has been developed (and prioritized via a risk analysis process), it is appropriate to turn to tools to help automate the testing to the extent possible. Traditional penetration testing tools such as vulnerability scanners will no doubt fall short of being highly useful, but can be a reasonable starting point – and, without a doubt, can help point out human errors in the configuration of a deployment environment.

It is likely that much of the testing will need to be executed manually or with custom built scripts, prototypes, and such.

[1] For a more detailed discussion on adapting penetration testing to the needs of software developers, see the author's article at http://BuildSecurityIn.us-cert.gov.

However, a vital component of penetration testing lies in demonstrating the extent and severity of discovered weaknesses. Some penetration testing tools such as Metasploit can be useful in developing and demonstrating custom written exploits for these defects.

Code review

The good news in this area is that static source code analysis tools have come a long way since first being introduced around 1999. Today's commercial tools for analyzing source code for security defects is indeed highly capable and useful to the software developer.

The bad news is that it is all too often the case that development teams attempt to use these tools using much the same late lifecycle approach that they do for penetration testing. Indeed, many software developers even hire a team of consultants to scan their source code en masse after it has been fully developed. This dooms the tools and the process to failure.

Even the best static analysis tool available today still generates enormous amounts of findings on even a moderately sized source code tree the first time it is analyzed. This results in substantial amounts of time spent studying the results and trying to make sense of them – and decide how to proceed.

This is probably the most common mistake made in trying to use static source analysis tools, but it is far from the only one. Here are some tips that are worth consideration when trying these tools out.

- All at once. Don't expect miracles when throwing your entire source tree at a static analysis tool. You're bound to be overwhelmed with follow-on analysis and are more than likely to throw your arms into the air in despair.

  Instead, integrate the tool into your nightly build process. When developers check their code back into the source tree, make it mandatory to run the code through a static analyzer. That way, the developers will iteratively analyze their own code modules for problems while the development process progresses. Don't worry about re-analyzing the same code over and over; all the commercial tools on the market today are quite smart about tagging and tracking their findings and not "rediscovering" every time the tool is run.

  All of these tools work best iteratively than they do in analyzing entire huge source trees.

- IDE plug-ins. If your team uses a popular IDE for developing their code, you are probably in luck. Most of today's static analysis tools support a wide range of popular IDEs in the form of plug-ins that seamlessly integrate into the IDEs themselves.

  In most cases, the developer can invoke the code analyzer by pressing a single button (say, over lunch time or just prior to checking the code back into the CVS (or similar) tree. Once run, the tool's findings pop up throughout the source code navigator window whenever the developer views that portion of the code.

  This IDE integration can be enormously useful to the software developer, at least for those who have moved on from vi and EMACS.

- Management features. Most of today's static analysis tools include an "enterprise" deployment capability that includes a security manager server. The managers are generally used to track findings across entire development teams, visualize statistical reports, and to make and enforce policies.

The fact is that most of the management servers can be extremely useful in development teams. The policy engines, for example, are an excellent way of ensuring the big problems (e.g., cross-site scripting) are handled appropriately by experienced coders, and not just dismissed by the person running the analysis tool.

- Custom rules. Once again, most of the available tools provide the user with the ability to build customized rules to scan source with. In many enterprise development organizations, these custom rules grow to be about as important as the set that came with the tool from the vendor. With a bit of learning (see the discussion below on getting the most out of the tools), you can build a set of rules that are relevant to your APIs, libraries, naming conventions, etc., and ensure your developers adhere to them. This capability is enormously useful when used appropriately.

Application security testing

Although there are a myriad of other security-relevant tests that can be done against software, we'll focus on two additional test categories here: dynamic validation and risk-based testing. The bad news is that most of these tests require manual processes and/or customized test rigs that script the various tests. However, there is still plenty of room for automation or at least making use of tools that can provide valuable insight in the process.

A few such tool types, along with tips on how to use them, follow:

- Process analyzers. Process analysis tools can be invaluable at determining and validating the dynamic characteristics of a piece of software. They can determine for example, what files, threads, processes, registry keys, etc., are in use by any particular process as it runs. Generally, they provide a read-only interface to observing the software's behavior, but this can be highly useful at verifying the actions taken by the software being tested.

- Interposition tools. Where process analyzers are read-only "scopes" for observing an application's behavior, interposition tools are an active layer between an application and its environment or client. Commercial tools such as Holodeck interpose themselves between an application and its operating system and enable the tester to not just observe system calls and such, but to affect their return codes to observe how the software behaves. For web application, so-called *application proxies* can do the same thing by interposing themselves between the client browser and the web server being tested. These are enormously powerful tools for testing the implicit assumptions in software, such as over-reliance on client side scripting and the like.

Getting the most out of them

Irrespective of which tool or tools you end up using, it is without a doubt in your best interest to try to get the most of your investment. Even free or open source tools are expensive in the sense that they take time and energy to learn and use properly. Here are some tips to consider along those lines:

- Test scenarios. In many cases, your most difficult challenge won't be the tools themselves, but developing realistic test scenarios for the test team to execute. This is especially the case if it is the QA test team that does the security testing, since they've spent years learning how to test to functional specification and are often ill-prepared to seriously consider how software security can be broken. To this end, do consider involving your information security team along with the developers and testers to brainstorm and document valid, realistic, and topical test scenarios that will rigorously test your software against the threats it is most likely to face once deployed.

- Learn everything. Step number one in getting the most of any testing tool is to learn it inside and out. Go through the vendor documentation and experiment. Learn everything there is to learn about the tool. Believe it or not, many "shelfware" failures result from the users trying to get through a tool by nothing more than the *feel* of its GUI.

- Vendor training. Similarly, if your vendor provides it, seriously consider sending one or two key staff members to the vendor's own product training. It might well end up costing less in the long run, particularly for a complex tool, to do this than to self-learn and experiment.

- Designated key personnel. Consider designating a number of key employees as experts for each particular tool you use, once again especially the complex ones. Those key personnel should be available to help and mentor junior staff who have not gone through the vendor training. Among other things, the knowledge sharing will likely benefit the entire team, not just the key personnel.

- Vendor support. Here too, if your vendor provides support—perhaps via different tiers of expertise and availability—consider making use of it. It's likely to pay for itself in resolving difficult issues with the product and/or validating product bugs and such. If you do go this route, track your utilization of the support contract to verify you're getting your money's worth, but do at least give the vendor a fair chance. Their engineers often have more inside knowledge of a tool than anyone picking it up is likely to have without significant "time in cockpit" experience with the tool.

Conclusion

This paper has presented a quick overview of many of the issues faced in effectively using security-testing tools in a software development process. Clearly, there are many issues and it is quite easy to see how common mistakes get made. However, by carefully applying each tool and not expecting miracles from it, the tools can help security testers provide significant value to the software development team.