Secure Coding.
Practical steps to defend your web apps.

Copyright SANS Institute
Author Retains Full Rights

This paper is from the SANS Software Security site. Reposting is not permitted without express written permission.

Interested in learning more?
Check out the list of upcoming events offering "Defending Web Applications Security Essentials (DEV522)"
Although attacks on applications are a major source of security breaches (see Verizon DBIR, etc.), SANS research shows most organizations spend only a tiny percentage of their IT security budgets on application security, and that securing management buy-in and funding for application security programs are serious, ongoing challenges for maintaining successful application security programs.

Spending some time and money up front on application security can significantly reduce costs and risks later by reducing the number of weak points that could be exploited by attackers. The trade-off is that time and money spent on preventive security is time and money that you can’t spend accelerating the pace of your software delivery. You want to spend just enough to reduce security and operational risks to a level that is acceptable to your customers, partners, management and regulators. The magic is in determining the right balance between proactive control, reactive agility, tolerance for the risks facing your organization and selecting the right tools and practices to make your applications secure without slowing development.

Calculating overall risk to decide how much to spend on security means evaluating risks to the industry, the organization, its systems and its tolerance for risk in addition to calculating the cost of a catastrophic data loss and its aftermath.

In this paper we will focus more narrowly on the impact of application security on the end-to-end software development value chain. We’ll also look at ways to identify and balance cost and risk to help you decide which tools and practices are most practical and cost effective for your organization.

---

No. 1 Source of Breaches

Many surveys have named public-facing web apps as the leading source of data breaches, but they are not the only vulnerable types. The 2016 SANS State of Application Security survey found leading sources to be:

- Public-facing web applications: 41%
- Legacy applications: 30%
- Windows OS: 30%
- APIs developed in-house: 28%
- Applications hosted in the public cloud: 28%
- Business applications managed internally: 28%
- Custom applications: 26%
- Third-party open-source applications: 24%

---

Some organizations bet on their ability to catch security problems in production and fix them quickly, possibly even catching attackers in the act and closing a window of opportunity before anything bad can happen—instead of wasting valuable time up front. If they are fast enough, they can catch attackers and close the window on them before any real damage occurs.

If you work at one of these organizations, there are several reasons to reconsider this approach.

First, attacks start almost immediately. A recent study by the BBC found that a new server connected to the Internet will be identified and attacked in little more than an hour.5

Next, you have to catch attacks in progress—which is much harder to do than you think. You need to have strong monitoring capabilities, know what to look for, and be prepared to respond immediately when you see it—which is why most security breaches aren’t found until long after the attack actually occurred. Does it matter that you can get a fix out to a vulnerability in less than a day if it took six months to find out that an attacker had already exploited it?

And, as we’ll see in this paper, fixing security problems in production isn’t cheap, even for DevOps teams with automated delivery and deployment pipelines in place.

Fixing a simple SQL injection or XSS problem shouldn’t take long, if you know where to find the vulnerability and how to fix it. But if you didn’t bother to protect your app against common and dangerous attacks like XSS or SQL injection, you’ll also need to fill in hundreds of other holes.

Other problems are much, much harder to fix after the system is in production. If you made a fundamental mistake in architecture or requirements, you may have to go back and rewrite lots of code, which could force changes onto other systems and your users. Redesigning distributed identity management protocols, moving trust boundaries with outside systems and rewriting code in a safer language are just some examples of things you can’t afford to do in production.

Overall, it is easier, faster and cheaper to fix a mistake during development than it is to remediate a vulnerability post-production or clean up after a data breach. We’ll compare issues and costs at three points in the cycle: in development, in production and after a security breach.

**How Development Methodology Affects Security Costs**

Finding and fixing a mistake during early design and development reduces the cost of remediating a problem and any data breach it makes possible by limiting its *blast radius*. The costs of a mistake increase over time because security is a *cross-cutting concern* that can affect many different features—a single error, oversight or misunderstanding may force you to change a lot of code.

Several different factors determine how much this could actually cost, but one of the most important is how the team designs, develops and delivers software.

**Waterfall and Sequential Project Delivery**

In the classic waterfall model, software development moves through sequential steps: planning, requirements, design, code, test, acceptance and deployment. Waterfall teams spend extra time up front to understand and plan the work in detail. They freeze the specification and design, and handle new information (changes, new requirements and fixes) as exceptions.

As work is completed, it is handed off from business analysts to architects to designers, to coders and testers and finally to operations, with reviews and approvals at each point. Testing and acceptance are left to the end, which means that problems also turn up late in the process and can be very expensive to fix.

Each problem found late in testing needs to work its way back up and then down through the waterfall again. A mistake in requirements (for example, forgetting to include a security feature or privacy constraint) means code must be pushed upstream to requirements analysis and then down through all of the subsequent steps—design, coding, testing and acceptance—yet again. Along the way, specs, models and other documentation need to be updated.

Most of the studies that look at how expensive it is to fix vulnerabilities are still based on waterfall projects going back 20 or 30 years or even more, when development involved lots of paperwork and manual testing. In these cases the cost of making a change or fix late in a project could increase by 4x on small projects to 30x (using example data from a NIST study in 2002)\(^6\) and up to 100x in large enterprise systems.\(^7\)

---


\(^7\) [“The Real Cost of Change in Software Development,”](http://swreflections.blogspot.com/2013/09/the-real-cost-of-change-in-software.html)
While it's clear how and why costs increase over time in waterfall projects, using old data doesn't make sense today, when most teams are working in much more agile, iterative and automated ways.

**Agile and Lean: Iterative and Incremental Development**

Modern Agile teams work with the understanding they will never have enough information up front to make perfect decisions. Instead of trying to spec out and design the system in detail, they start simple and iterate. Teams deliver software as often as possible so they can get feedback from testers and users to validate their understanding of user requirements and their design decisions. Prototyping, experimenting, continuous testing and commitment to refining the prototype are all considered part of the development/design process.

Agile methods, such as Scrum and XP, and Lean models, such as Kanban, attempt to flatten the cost of making changes and fixes.

Instead of writing detailed specs and design models, Agile teams draft user stories and emphasize delivering working software over documentation. Changes and fixes are made by the team working together—no need to pass work back and forth among analysts, designers, coders and testers. As much as possible, the testing and build steps are automated so testing can be done early and often.

**DevOps and Continuous Delivery**

DevOps applies Agile ideas and practices to packaging, staging, releasing and deploying, and running the system, thereby optimizing and automating the full path from design to production.

DevOps drives the cost of making changes and fixes even lower by automating and streamlining the important "last mile" to production with Continuous Delivery and Continuous Deployment and by minimizing and simplifying hand-offs between development and operations. As in Agile development, there is less paper and bureaucracy, and more face-to-face collaboration and joint problem-solving. DevOps teams also manage infrastructure in code, making it cleaner and faster to identify security vulnerabilities and correct them, compared to trying to patch and update an enterprise legacy system.
Fixing Security Problems Found in Production

Because most teams today are working in Agile or working toward DevOps environments, we will assume that a much flatter cost of change curve is realistic for them.

Regardless of how the team develops and delivers software, the costs to fix security problems increase once a system is in production, whether these issues are found through monitoring, vulnerability scanning or threat intelligence or by a bug bounty hunter or a customer. This cost increase is because there is a lot more involved in fixing a security problem in production than making a software change.

Cost Factors

The costs of finding and fixing a security vulnerability in production include:

1. Understanding the problem and its impact, which could involve pulling security and engineering team members into incident response
2. Investigating to determine whether this vulnerability was exploited, which could involve bringing in forensics experts
3. Interrupting engineers and disrupting whatever they were working on, causing them to lose focus
4. Understanding how to fix the problem and how much cost and risk are involved in making the change
5. Working with the security team, compliance and management to decide whether the problem is worth fixing now or later—or not at all
6. Designing and implementing the fix or downloading a patch
7. Building a new version of the code
8. Testing to make sure you fixed the problem correctly and to ensure you didn’t break anything else accidentally
9. Getting the change or patch approved and scheduled, including understanding and dealing with any operational dependencies
10. Deploying the new code to production
11. Monitoring the system to make sure you fixed the problem and to ensure you didn’t break anything else accidentally
12. Tracking all this work for compliance and governance purposes
Complicating Factors

These costs depend on several factors:

1. How clear the problem is and how well you understand it, which depends on how the problem was reported, how much information was given to the team, how much security knowledge the developer has and how well the developer understands the code (that is, whether they’ve worked on this code before).

2. How easy it is to build the code. Are the build steps documented or automated? When was the last time the code was built successfully?

3. How hard the code is to test. Does testing need to be handed off to an independent test team, or can developers rely on automated regression tests in Continuous Integration (CI) or Continuous Delivery (CD)?

4. What you have to do to prove the fix is good. Do you need to add a unit test, rescan with a tool or submit back to an auditor or a bounty hunter?

5. The costs to coordinate, deploy and roll out system changes.

6. Governance and compliance requirements for tracking, reporting and follow-up.

The actual coding work involved can range from less than half an hour for fixing a simple SQL injection bug or a mistake in error handling, or adding a bounds check or missing ACL check; to an hour or so for downloading a patch to a third-party library; to up to several hours or days to fix complex logic or make fundamental changes to shared services or key workflows. The coding cost also depends on whether you decide to make a quick and dirty patch to address the immediate problem or to look deeper and make changes to ensure the code is safer in the long term—or both, by pushing out a quick fix immediately and scheduling a more comprehensive and cleaner implementation later.

Most of the time and cost involved in fixing a vulnerability is not in coding, but in building and testing the fix and getting it deployed. Fortunately, as we’ve seen, modern Agile and DevOps practices optimize for all of this.

You also need to factor in more costs if you don’t catch the problem in time—for example, if an attacker has already successfully exploited a vulnerability before you were

---

See research from Denim Group from 2012 on how much it costs to fix vulnerabilities: https://software-security.sans.org/blog/2012/11/05/ask-the-expert-dan-cornell
able to catch it and patch it.

In 2016, RAND Corp. looked at 12,000 security events, including data breaches and privacy violations, going back over a 10-year period. They found that the direct costs of a typical security incident, including forensics, recovery and litigation, were around $200,000 per incident.⁹

But this does not take into account lost revenues due to brand damage or follow-on costs imposed by regulators, customers and other stakeholders. When you add in these indirect costs, the total cost of dealing with a data breach or other serious incident skyrockets, as we see in another study published in 2017 by the Ponemon Institute.¹⁰ They found that the total cost of a security incident (excluding high-profile mega-breaches like those at Target or Yahoo) came to $3.6 million per incident, with indirect costs considered. Total costs can run much higher depending on the amount of data lost or compromised, and in highly regulated industries such as financial services or healthcare, where noncompliance carries hefty penalties plus fines for each customer record compromised.¹¹ The potential costs of violating the EU’s new General Data Protection Regulation (GDPR) privacy laws are even more serious.¹²

---


We've gone through the evidence to prove that it makes good sense to spend some time and money during development, in order to save more time and money later. But where will you get the best bang for your buck? Should you spend money on people or tools, and how much money do you need to spend to get meaningful results? Which tools make most sense for your situation?

These decisions will depend on the variables listed in Table 1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Questions to Ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>How much money do you have to spend?</td>
</tr>
<tr>
<td></td>
<td>How much have you already spent?</td>
</tr>
<tr>
<td></td>
<td>Whose money are you spending? (Different sponsors have different priorities.)</td>
</tr>
<tr>
<td>Experience</td>
<td>What successes have you had? What failures?</td>
</tr>
<tr>
<td>Size</td>
<td>What is the size of organization, system, data and project?</td>
</tr>
<tr>
<td>Compliance</td>
<td>What are the legal and compliance constraints? (Prescriptive regulations like PCI DSS dictate specific practices and controls.)</td>
</tr>
<tr>
<td>Culture</td>
<td>What is the organizational/engineering culture?</td>
</tr>
<tr>
<td></td>
<td>What is the tolerance for risk?</td>
</tr>
<tr>
<td>Engineering</td>
<td>What methodology/ approach is needed?</td>
</tr>
<tr>
<td></td>
<td>What maturity and technical capabilities need to be considered?</td>
</tr>
<tr>
<td>ROI Term</td>
<td>How much time do you have to show results?</td>
</tr>
</tbody>
</table>

Your decision making will be easier if you build a simple model, evaluating tools and practices based on their cost, their effectiveness at reducing risk, and their net effect on your product development or service delivery organization’s ability to get meaningful work done.
**Costs**

Implementation costs and ongoing costs of implementing a tool or adopting a manual practice include the following:

1. Finding and selecting a tool (including running a POC), or interviewing and hiring people
2. License/service costs (if not open source) for technology
3. Labor costs for manual testing and reviews
4. Implementation project overhead and management costs
5. Integration costs of wiring into tool chains and workflows, including CI/CD, vulnerability management and compliance reporting systems
6. Training costs and evangelism (selling to developers and managers)
7. Support costs
8. Administrative and compliance costs

**Effectiveness**

How much will this reduce security risk to your organization, and how much will this save you on long-term costs?

This is about catching and preventing real problems as early as possible: high-fidelity, high-risk security vulnerabilities that all stakeholders—security, compliance, engineering and management—agree are worth fixing, based on how easily these problems were to discover and exploit, and their potential impact. How high or low this bar gets set will depend on the organization’s (and the system’s) risk profile, the compliance environment and the engineering team’s maturity.

You measure this in two different ways:

1. **Number of real problems caught.** Scanning tools and pen tests make this straightforward because they generally rank findings by severity using standard scoring models, such as CVSS\(^{13}\) or OWASP’s Top 10 Risks,\(^{14}\) and provide additional context to help decide whether a problem needs to be fixed. You can use the same criteria to prioritize problems found in code reviews or design reviews or manual testing.

2. **Number of real problems missed.** You do this by tracking problems found by other techniques and in production.

---

\(^{13}\) “Scores and Calculators,” [www.first.org/cvss/scores](http://www.first.org/cvss/scores)

You will get some important insights by tracking the number of security problems found, and how or when you found them. Counting the number of real problems found in scanning or reviews doesn’t just tell you how much return you are getting from a specific tool or practice; it also tells you how many problems are escaping from other tools, tests or reviews.

Catching a problem early in reviews, such as a missed requirement, or choosing to use (or not use) a framework or to move a trust boundary, could save you from having to deal with many more problems later. This will show up over time as you look at the number of problems found by other tools or reviews, or in production.

Consistency is an important factor in determining effectiveness. Automated scanning and automated testing provide more consistent results than manual reviews and testing, which are highly dependent on people’s skills, experience, attention and availability, which can vary widely.

**Efficiency**

Efficiency looks at how much work is involved to find real problems, including the drag on engineering teams and their ability to deliver. How many hours or days per year will engineers need to spend scanning or reviewing or testing, or waiting for results to come back and working through them, and tracking all of this instead of delivering working software? What is the total cost to the organization, including opportunity costs? You can answer these questions through value-stream analysis.

Time is money. Wasted time is wasted money. Lean/Agile and DevOps teams will be watching out for waste as they optimize cycle time to delivery, identifying and solving bottlenecks and eliminating overhead and rework and delays. Anything that gets in the way of delivering direct value to the organization needs to be evaluated, justified and optimized, including security and compliance/risk management controls.
Complicating Factors

The factors to be considered in this include:

1. **Scalability**—How much of the system can be covered consistently. Automated scanning is much more scalable than manual reviews and testing, and scanners never get tired of going over the same code and looking for the same problems.

2. **Friction (delay on feedback)**—How long it takes to provide meaningful feedback to the developer (frequency and speed of checks).

3. **Quality of feedback**—Context and guidance offered to developers, and how clear the problem is to a developer and how to deal with it.

4. **Accuracy and validity**—Ratio of true positives to false positives or other noise that you don’t care about fixing or don’t need to fix right now.

5. **Flow**—Impact on developer experience and flow. How seamlessly and naturally does this fit into the way developers think and work on a day-to-day basis? How does it integrate into engineering tool chains or workflows?

Simple Return on Investment Model

Compare the return on investment for different practices or tools using this simple model:

\[
\text{Effectiveness} / (\text{Cost} + \text{Efficiency})
\]

Or in other words:

\[
\text{Number of high-impact problems caught} / (\text{Implementation costs} + \text{Ongoing costs} + \text{Indirect drag and opportunity costs caused by delays})
\]

Automated software component analysis (for example, scanning build dependencies or container images for known vulnerabilities) can be added relatively easily into the build without changing how the team works. Total costs are modest: an upfront license for the tool (which could be open source) and integration into the build, which should take a day or less for a simple system or a few days for a large multistep build; plus some time to review the initial findings and set a risk baseline. From that point, feedback is immediate and straightforward on each build: The tools show where vulnerabilities were found and how serious they are, with few false positives. You can easily measure the number of real problems found (versus noise) and calculate how much work is required to remediate them.
Manual assessments like threat modeling require more effort from the team: They need to make time to do reviews, and take them seriously. Costs depend on how often reviews are done and how many people are involved. Assuming that teams are following Scrum, for example, a threat model or design review scheduled in each one- or two-week sprint could involve five to seven developers, the product owner, the Scrum Master and a security expert. The process would also require some time to prepare and follow up. Even if you timebox each review to one hour, the ongoing commitment required is significant. Results will be inconsistent—because they depend heavily on reviewer ability and availability and the quality of information that reviewers are given—and difficult to quantify.
In this section we assess some common secure SDLC practices and tools to determine which have the best payoffs. We start with techniques that can be used early in the value stream, which is where we can save the most time and money.

Finding security vulnerabilities requires close examination of the big picture and the small. First, the big picture: finding vulnerabilities and compliance issues in application architecture and design (see Table 2).

<table>
<thead>
<tr>
<th>Table 2. Threat Modeling/Architecture Risk Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>Review costs depend on the extent of team involvement and outside experts. May also include license costs for application security requirement and threat-management tools. Effective reviews require smart people who understand system architecture, the problem domain and the operational environment. They also need to understand security and privacy and compliance risks and threats, and how to manage them. These people are hard to find and in high demand. You may need to invest in training or bring in expensive consultants.</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
</tr>
<tr>
<td>A rule of thumb is that roughly half of all security vulnerabilities are design flaws (versus coding or implementation defects), some of which can be caught by threat modeling or architecture risk analysis. Reviews can identify missed technical requirements or controls, mistakes in trust assumptions and exposure of sensitive data, and high-risk features that need careful testing. It's possible to prevent serious security problems altogether by constraining the team’s solution space and providing them with a choice of proven, secure patterns and vetted technologies—what Netflix calls “paving the road ahead” for the engineering team.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td>To make good decisions, reviewers need good information, including requirements and models to work from, and people who can walk through the design and answer questions. Reviewers with the necessary experience and skills are hard to find and are not always available to the team when you need them, which typically causes expensive delays or forces teams to compromise in order to stay on schedule by holding reviews without the right people in the room. Reviews can easily be added as a gate in waterfall projects, before the design is ready to be handed off for coding, using the specifications and design models created along the way. But how do you make this work in highly iterative Agile/DevOps environments where the design is never done and where teams focus on delivering “working software over documentation?” Conducting design reviews on an incremental, continuous basis will spread reviewers extremely thin.</td>
</tr>
</tbody>
</table>
After the design is approved comes the more painstaking task of examining individual lines of code (see Table 3).

<table>
<thead>
<tr>
<th>Table 3. Manual Code Reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>Costs depend on the approach: pair programming or lightweight peer reviews, formal inspections, or external code audit (most expensive). Costs also depend on the number of reviewers invited, how often they are asked to review code, and the amount of evidence needed for appropriate governance and compliance. Manual costs: the time developers spend in reviewing each other's code and dealing with reviews on their code, including asking and answering questions, and responding to review findings. At Microsoft, for example, developers spend roughly six hours per week reviewing each other's code.15 Other costs: license/implementation costs for code review managers like Gerrit, or Review Board or Crucible. Training: teaching developers secure and defensive coding so they know what to look for. Developers may need additional training or mentoring on how to effectively review code.</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
</tr>
<tr>
<td>Results depend heavily on several factors: the experience, knowledge and focus of the reviewer(s); the number of reviewers; how much code they are asked to review; and the amount of time available. Reviewers can find common mistakes in coding, unsafe practices, mistakes in business logic and errors in design. This depends on the reviewer's familiarity with the code base and conventions, understanding of the language and the readability and complexity of the code.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td>Friction/delay. It can take days for a reviewer to come back with findings and then more time to go back and forth to understand and respond to findings and questions. Drag on reviewers. Reviewing code takes them away from other useful work. Use of time. It takes time and reinforcement for code review practices to become a habit.</td>
</tr>
</tbody>
</table>

Automated tests can save time by identifying app requirements, configurations and initial states (see Table 4).

| Cost | License/service cost (although there are good open source tools for some languages).
|      | Implementation and integration into build chains can be tricky. Requires someone who understands your build environment well to set up correctly. |
| Effectiveness | Most apps include a lot of third-party/OSS code: frameworks, components, plug-ins and container images. You inherit vulnerabilities from this code. 
|      | Popular libraries with serious vulnerabilities create high-risk conditions: automated scans, weaponized attacks causing widespread damage at low cost to attackers. 
|      | SCA can find problems automatically at build time. Availability depends on language and packaging approach. Accuracy/completeness depends on quality of vulnerability databases used. |
| Efficiency | Integrates into the build chain: automatic scan and alert, or fail build/deploy. With some more advanced tools, you can block downloads of outside code with known problems (like a firewall). 
|      | Upgrading dependencies may not be as simple as downloading a patch. It may require moving to a fork (incompatible), or you might be forced to fork and fix the code yourself. |
Static testing checks app source code, byte code and binaries while apps are not running to identify vulnerabilities (see Table 5).

<table>
<thead>
<tr>
<th>Table 5. Automated Static Application Security Testing (SAST)</th>
</tr>
</thead>
</table>
| **Cost** | License/service cost (may depend on size of team or size of code base).  
Implementation and integration costs (integrating into IDE and build chain); configuring, testing, and tuning tools/engines; making sure code is covered properly.  
Explaining to developers and convincing them to use it. |
| **Effectiveness** | Good at finding common coding mistakes, including buffer and memory management problems in C/C++ and pointer handling, data type and arithmetic mistakes, unsafe functions, embedded credentials, common mistakes in crypto, injection vulnerabilities, copy and paste errors, information leakage and holes in error handling.  
Coverage depends on:  
What code is included (integration into build or binaries, third-party/OSS libraries)  
How the scanning engine works (Different scanners look for different problems, and some scanners are better at finding certain problems than others.)  
The number of scanners being used (Studies from NIST, OWASP show you should use multiple scanners to get high coverage.)  
Support for your language(s) and framework(s)  
The rule set (Do you include slow, noisy rules?)  
Whether you are scanning incrementally (fast) or full (slow)  
How often you scan |
| **Efficiency** | Depends on scanning method—by a central factory (slow feedback) or integrated into developer build chain/workflows. Some tools can catch problems in the developer's IDE as they make changes or can run incrementally in CI/CD, while other tools require full/deep scans which can take days on large code bases.  
Quality of feedback. Good tools will tell you what the problem is, where it is, why it is important and how to fix it.  
Need to trade off completeness with accuracy. Some rules are noisy and need to be tuned, or reviewed and triaged so that they don't waste developer time. |
Dynamic Application Security Testing (DAST)

Dynamic testing looks for vulnerabilities while the app is running, especially in HTTP and HTML interfaces. It is also used to detect issues in RPC or other non-Web interfaces and data format issues (see Table 6).

<table>
<thead>
<tr>
<th>Table 6. Dynamic Application Security Testing (DAST)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>License/service plus implementation costs.</td>
</tr>
<tr>
<td>Upfront time needed to set up, configure and train the tool to understand the application's attack surface and integrate into functional test suites.</td>
</tr>
<tr>
<td>Integration costs: integrating into CI/CD or automated testing cycles.</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
</tr>
<tr>
<td>Good for finding certain injection vulnerabilities, cross-site request forgery (CSRF), common configuration mistakes, information leaks and mistakes in error handling.</td>
</tr>
<tr>
<td>Limited to web and mobile apps; some support for scanning REST APIs.</td>
</tr>
<tr>
<td>Many tools have difficulties scanning complex UIs with advanced JavaScript etc.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td>Results are more difficult to understand and use than white box techniques, and there tends to be more noise.</td>
</tr>
<tr>
<td>Can be difficult to automate in CI/CD. Dynamic scans can hang or fail.</td>
</tr>
<tr>
<td>Time to scan large systems—can take hours/days, not suitable for CI/CD and needs to be done out of band.</td>
</tr>
</tbody>
</table>

Testers using many of the same techniques as attackers can identify many of the same problems before they can be exploited after an app is deployed (see Table 7).

<table>
<thead>
<tr>
<th>Table 7. Penetration Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>Costs (and effectiveness) depend highly on the tester's skills, knowledge and methodology, and on the time they are given to do the testing.</td>
</tr>
<tr>
<td>Find/hire testers, set up system with test users/data, and brief testers on how the system works.</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
</tr>
<tr>
<td>Depends on the approach/strategy: low-cost compliance scans (highly structured checkbox approach) versus expert tiger team (deep, comprehensive, not done until they own you) and whether the engagement is time-limited or goal-based.</td>
</tr>
<tr>
<td>Realistic. A good pen tester will find many of the same problems an attacker would, including runtime configuration mistakes and business logic bugs.</td>
</tr>
<tr>
<td>Diminishing returns will set in over time, where a pen tester's approach and knowledge set plateaus.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td>Late-stage testing activity. You need to set up a working system to be pen tested, which means feedback comes toward the end of delivery.</td>
</tr>
<tr>
<td>Long feedback cycle for each test. You need to coordinate and set up the test, run the test, review results (back and forth with the pen tester to qualify and understand findings) and get results out of reports into the team's backlog.</td>
</tr>
<tr>
<td>Developers must be involved to assist pen testers, provide context and review results.</td>
</tr>
</tbody>
</table>
A more extended version of penetration testing is the practice of paying researchers or developers to find vulnerabilities in code and tell the publisher about them (see Table 8).

<table>
<thead>
<tr>
<th>Table 8. Bug Bounties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
</tbody>
</table>

Making Informed Trade-Off Decisions

Simple ROI models based on “cost to fix” arguments are no longer sufficient to justify investing in application security up front. You won’t scare today’s Agile or DevOps teams into changing how they work by using old data from waterfall projects—these teams are already riding a much faster change cycle, relying on automated build and deployment chains to push changes out continuously and cheaply.

But we’ve also seen that not taking responsible steps to reduce risks in development can lead to serious and expensive problems later in production, because of the amplified blast radius of early mistakes.

We need to look instead at the complete picture of cost and risk for the software delivery value stream and identify how and where we can reduce risks in the best way possible. This needs to account for not only direct costs, but also how each decision interferes with the flow of delivery, and trade these decisions off against how much they measurably reduce security risks.

Adding expensive or inefficient security controls early in development—in an attempt to reduce the potential cost of future changes—can backfire badly. Every check you add, every dollar you spend on risk management, impacts the flow of value. You need to continuously evaluate and justify these decisions and ensure you are getting the best return on investment possible. Build this into engineering measurement and feedback loops so you are constantly assessing and improving your results.

There are better or worse answers for your organization, based on your technology and operational environment, your engineering teams and how they work.

Thinking and talking in terms of end-to-end value streams and ROI—and framing your application security investment decisions based on ROI—instead of focusing solely on risks and compliance will help align security priorities with the rest of the organization and help people make decisions that everyone understands and owns together.
Jim Bird, SANS analyst and co-author of DEV534 Secure DevOps, is an active contributor to the Open Web Application Security Project (OWASP) and a popular blogger on agile development, DevOps and software security at his blog, “Building Real Software.” He is the CTO of a major U.S.-based institutional trading service, where he is responsible for managing the company’s technology organization and information security program. Jim is an experienced software development professional and IT manager, having worked on high-integrity and high-reliability systems at stock exchanges and banks in more than 30 countries. He holds PMP, PMI-ACP, CSM, SCPM and ITIL certifications.

SANS would like to thank the sponsor of this whitepaper:

[Logo of Veracode]
<table>
<thead>
<tr>
<th>Course Description</th>
<th>Location</th>
<th>Dates</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univ. of California - Davis - DEV522: Defending Web Applications Security Essentials</td>
<td>Online Only, CA</td>
<td>May 11, 2020 - May 16, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>SANS Network Security 2020</td>
<td>Las Vegas, NV</td>
<td>Sep 20, 2020 - Sep 27, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Online</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s Only</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>