SYNOPSYS®

7 Myths of Software Security

TABLE OF CONTENTS

Page 3: Introduction

Page 4: **Myth 1:** Perimeter Security Can Secure Your Applications

Page 6: **Myth 2:** A Tool Is All You Need For Software Security

Page 9: **Myth 3:** Penetration Testing Solves Everything

Page 9: **Myth 4:** Software Security Is a Cryptography Problem

Page 12: **Myth 5:** It's All About Finding Bugs In Your Code

Page 15: **Myth 6:** Security Should Be Solved By Developers

Page 17: **Myth 7:** Only High-Risk Applications Need To Be Secured

Introduction

As you probably well know, new technologies are moving at incredible speeds these days. That's why building secure software should be a top priority in your organization. As more software is created, more vulnerabilities are also created. As these vulnerabilities (a.k.a. the broken stuff) are created, attackers end up with a better chance at finding and exploiting them to penetrate your system—cracking into software full of sensitive data for malicious purposes. In the beginning, perimeter security was invented to protect this broken stuff from bad people. But, more importantly, why is the stuff broken in the first place?

These seven software security myths are common misconceptions about software security best practices. These myths explore how software security initiatives should work, and aren't simply about how to secure a particular application.

Perimeter security can secure your applications

A tool is all you need for software security

Penetration testing solves everything

Software security is a cryptography problem

Software security is only about finding bugs in your code

Software security should be solved by developers

Only high-risk applications need to be secured

Perimeter Security Can Secure Your Applications

How well does the perimeter security approach secure applications?

Perimeter security was designed to protect an internal network from the mysterious unknown of countless malicious users by selectively stopping network traffic coming in and out of the theoretically-protected network. Perimeter security has evolved over the years to include firewalls (and the extremely limited Web application firewalls (WAFs)), security information and event management (SIEM) products, and products that monitor the operating environment in real time.

While these systems are a worthy investment and can effectively shield the hustle and bustle of creativity and innovation within your network, they do nothing to secure the actual software you rely on. Perimeter security protects the broken stuff from the bad people with a thing placed at the perimeter.

Perimeter security originated in feudal times

Today's computer and network security mechanisms are like the city walls, moats, and drawbridges of feudal times. These perimeter security measures were implemented to deter and block disturbances from interfering with the goings-on inside. At one point, this may have been an effective way to defend against isolated attacks mounted on horseback. Once the attacker was spotted, all they had to do was raise the drawbridge to deny entrance.

While perimeter security is a good basic security precaution, a few things have changed since the heydays of the feudal system. As attack strategies have become more advanced in the past, say, 500+ years, a moat, city wall, and drawbridge are no competition for those who want to break in to collect the big prize that lies within the perimeter. Today's attackers have access to things like predator drones and laser-guided missiles! The modern castle requires quite a bit more protection against attacks these days than a moat and drawbridge of years past.

Feudal systems are a thing of the past…for a reason

Due to more advanced threats, we must make sure our modern-day castles—or should we say data, software, and networks—are protected by more than just perimeter security measures. More importantly, instead of securing broken software against attack, why don't we just build software that's not broken? That's what software security is all about; building security into your software as it is being developed. Take measures to build secure

That's what software security is all about; building security into your software as it is being developed.

software throughout the software development lifecycle (SDLC).

Building secure software means arming developers with tools and training, reviewing software architecture for flaws, checking code for bugs, and performing real security testing before release.

The ultimate goal is to drill down into resolving the vulnerabilities and build things properly from a security perspective. When discussing information security, a firewall is still useful once the software is secure, and should most definitely be deployed. It's your organization's front line of defense. It's basic insurance.

Perimeter security as an all-powerful answer? Not even close.

The biggest challenge in security over the past two decades has been the dissolution of the perimeter. Massively distributed applications take advantage of the cloud's efficiency; in doing so, these applications are working diligently to eradicate perimeters. Firewalls, WAFs, and SEIMs are very difficult to deploy and maintain effectively without a perimeter. As the cloud becomes more utilized, perimeter security will become less effective, just as the antiquated feudal security measures. Additionally, protecting non-broken stuff from intruders is a better strategy, from a network security perspective, than to be in the position of protecting broken stuff.

Creating software without taking measures to keep your internal data, and the data of your users, safe from intrusion defies the best practices of software development and software security. But what measures are companies throughout various industries taking to ensure their data is safe? The Building Security In Maturity Model (BSIMM) is working to answer this question by measuring software security initiatives of 67 organizations throughout 12 industries. How does your organization measure up?

Perimeter security as an all-powerful solution is indeed a myth. While it acts as an organization's first line of defense, it's not the one true answer to security. After all, perimeter security isn't protecting the software an organization counts on. It's just the first of seven common misconceptions that are often held as gospel within software security best practices.

The biggest challenge in security over the past two decades has been the dissolution of the perimeter.

MYTH 2

A Tool Is All You Need For Software Security

All software projects produce at least one common artifact—code. This source code is the number one software security touchpoint your organization should address when strategizing a software security initiative (SSI). We've made great strides in the last 15 years building technology to find some types of security defects in code. At the code level, the security focus is on implementation bugs, especially those that can be discovered by static analysis tools.

This brings us to the topic of our second software security myth. Tools have played a major role in the acceleration of finding and resolving many defects in code. But are they the key to solving security threats and vulnerabilities?

A history of security tools

To look ahead, we must first take a step back in time. Let's take a look at where security tools got their start.

Black box testing tools were the earliest approach for simple protocols (such as HTTP). Dynamically scanning a Web app for known problems using a black box approach is both affordable and appealing. However, generalizing to other protocols isn't feasible using this approach.

Next came the idea of looking at the code itself with static analysis tools (also called source code analyzers). These tools examine the text of a program statically, without attempting to execute it, by looking for a fixed set of patterns or rules in the code. Theoretically they can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be challenging.

Static analysis tools are capable of scouring a program's code with less fuss than dynamic analysis, which requires running the code. Static analysis also has the potential to be applied before a program reaches a level of completion at which testing can be meaningfully performed.

The promise of static analysis is to identify common coding problems automatically, before a problem is released. Technology for automating code review has improved vastly since the days of ITS4. Static analysis capability is now available worldwide through IBM, HP, Synopsys, and other firms.

© 2017 Synopsys, Inc. | www.synopsys.com |

Today, there is a combination of tools blending dynamic and static analysis in many interesting ways (albeit mostly for HTTP).

The best a code review can uncover is about 50% of security vulnerabilities.

The problem: Tools aren't a catch all solution

Although more advanced tools allow new rules to be added over time, if a rule hasn't yet been written, the tool will never find that problem. When it comes to security, what you don't know could very likely hurt you. Security ignorance truly isn't bliss.

Further, these tools can have an absolutely limited impact. Black box security testing only works for Web applications because HTTP protocol is stateless and simple. Code review tools only look for bugs in code written in certain programming languages. Code review tools also don't understand the context of the application for which the code is being written.

Doing code review is an extremely useful activity, however, since this kind of review can only identify bugs, the best a code review can uncover is about 50% of security vulnerabilities. The other 50% are architectural flaws which are very difficult (downright impossible) to identify from code.

Software security should leverage tools and automation whenever possible, but tools alone simply do not solve the problem. The old software testing adage applies in security as well as it did in quality assurance:

A fool with a tool is still a fool.

This adage leads into another weakness within secure development—the fact that security isn't yet a standard part of the programming curriculum. You can't really blame programmers who introduce security problems into their software if nobody ever told them what to avoid or how to build secure software. Not to mention that programming languages weren't designed with security in mind. Unintentional (mis)use of various functions built into these languages leads to very common and often exploited vulnerabilities.

The problem of polar implementation

Integrate software security testing tools and automation into your SSI for reasons of efficiency and scale. But do not confuse tool use with an SSI. An SSI uses tools to enable efficiency and scale of a good strategy, but does not devolve to only using tools, as many firms tend to do.

On the other hand, manual auditing is very time consuming and to do it effectively, human code auditors must first know how security vulnerabilities look before they can rigorously examine code. Static analysis tools

compare favorably to manual audits because they're faster, which means they can evaluate programs more frequently, they encapsulate security knowledge in ways that don't require the tool operator to have the same level of security expertise as a human auditor.

The solution: A happy hybrid approach

Use the tools. They expedite processes and eliminate the low-hanging fruit vulnerabilities. But don't underestimate the effectiveness of humans. Output delivered by static analysis tools require human evaluation. There's no way for a tool to know which problems are more or less important to the organization. There's no way to avoid sorting through the output, prioritizing which issues should be fixed now, and which ones carry an acceptable level of risk.

And remember, the tools really only tackle 50% of the problems. Static analysis tools can find bugs in the deep, dark corners of code, but they aren't designed to critique design. A comprehensive approach to software security involves holistically combining both code review and architectural analysis.

Tools also suffer from false negatives (the program contains bugs that the tool doesn't report) and false positives (tools report bugs that the program doesn't actually contain). False positives can cause irritation to analysts who are responsible for sorting through them, but false negatives are more dangerous because they lead to a false sense of security.

Maintain best practices

So, is it true that a tool is all you need for software security? No.

While tools (such as those that perform code review) are a proactive resource in secure development, they're not a sufficient stand-alone practice for achieving secure software. A comprehensive approach of tools and architectural review conducted by security experts is the only approach to achieving the most secure software.

Penetration Testing Solves Everything

Security testing is important. Conducting specialized penetration tests at the end of the software development lifecycle (SDLC) can be a rewarding security activity for your organization. Penetration testing is, after all, the most frequently and commonly applied of all software security practices. But, this isn't necessarily a good thing.

This is why penetration testing makes the list as our third myth of software security. Just like a tool can't solve the software security problem by itself, neither can penetration testing. Let's kick things off with an exploration of the two main reasons why penetration testing by itself isn't a solution to the software security problem.

The misapplied scenario

One element that's critical to the effectiveness of penetration testing involves who carries out the testing. Be very wary of "reformed hackers" whose only claim to being reformed is the fact that they told you they were reformed.

Fact is, you can't validate results of a test you don't understand. If a reformed hacker turns out to be malicious, you're in trouble. To give you an example: an organization hires a group of reformed hackers. You know they're reformed because they told you they were. You give them a set time period (let's say one week) to perform a penetration test. At the end of the week, the reformed hackers have discovered five bugs. They tell you about four of them. Of the four you know about, only one is easy to fix, but your team manages to fix two. The other two—or was that three?—must wait. And you never even heard about one of them.

And that, ladies and gentlemen, is an example of how not to approach penetration testing.

Testing for positives and negatives

Be aware that there is a difference between network penetration tests and application/software-facing penetration tests. Additionally, a majority of software security defects and vulnerabilities don't directly relate to security functionality. Rather, security issues often involve unexpected (albeit intentional) misuse of an application discovered by an attacker. If we characterize functional testing as "testing for positives" (as in verifying that a feature performs a specific task as it's intended), then penetration is in some sense "testing for negatives." A security tester must probe into the security risks (driven by abuse cases and architectural risks) in order to determine how the system responds to an attack.

Passing a penetration test leaves organizations with a false sense of security.

Testing for a negative poses a far greater challenge than verifying a positive. It's a simple task to test whether a feature works or not. It's much more difficult to show whether or not a system is secure enough under malicious attack. How many tests do you do before you give up and declare "secure enough"?

If negative tests don't uncover any faults, this is only proof that no faults occur under particular test conditions, and by no means proves that no faults exist. "Passing" a software penetration test provides little assurance that an application is secure enough to withstand an attack. Many organizations misunderstand this point. As a result, passing a penetration test leaves organizations with a false sense of security. Don't forget to focus on risk and manage that appropriately.

The badness-ometer

We'd all love to have a security meter that tells us if our software is secure. But, there's no such thing as a security meter. There's only a badness-ometer. How's it work, you ask? First, you take smarts built into a hacker who can perform reasonable black box tests, and you make a collection of those tests and put them in a can. Next, take the can of black box tests (that, incidentally, don't know anything about software) and run them against program A. Imagine the canned tests break program A. What did you learn about program A? It's bad. So bad that a canned test that knows nothing about program A can break it!

You use the same canned black box tests against another program, call it B. If the canned tests don't break program B, what do you learn about program B? Not much. You ran some tests and they didn't find anything. Does this mean program B is secure? Nope. It means you ran a single set of tests, which for any number of reasons, or no reason at all, can't break program B.

The mis-timed scenario (a question of economics)

Problems are more expensive to fix at the end of the lifecycle. Economics dictates finding defects as early as you possibly can. Have a flaw in your idea? Redesign it in your mind (presumably free). Have a bug in your code? Find it while it's being typed in, not later during the build process. Want to find vulnerabilities in your software? Why wait until it's shipped?

Penetration testing is about testing a system in its final production environment. As such, it's best suited to probing configuration problems and other environmental factors that deeply impact software security. Driving tests that concentrate on other factors such as some knowledge of risk analysis results is the most effective approach.

One reason why so many organizations turn to penetration testing first is that it's an attractive late-lifecycle activity that can be carried out in an outside-in manner. Like the canned test, in some instances, you don't really need to know that much about the software being tested. As a result, basic penetration testing is a common activity that can be carried out on a completed application, under time constraints specified by the operations team, to fill a security testing checkbox at the end of the SDLC. Of course, fixing things at this stage of the game is, more often than not, prohibitively expensive (and in some cases involves configuration Band-Aids rather than construction-based cures).

BOTTOM LINE:

Outside-in testing is great as long as it's not the only testing you do.

The importance of architecture risk analysis and code review before penetration testing

Organizations that fail to integrate security throughout the development process are often unpleasantly surprised to find that their software suffers from systemic faults both at the design level and in the implementation. In many cases, the defects uncovered in penetration testing could have been found more easily through other techniques earlier in the lifecycle.

Testers who use architecture analysis results to direct their work often reap great benefit. For instance, if architecture analysis concludes that the security of the system hinges on transactions being atomic, then torn transactions become a primary target in adversarial testing. Adversarial tests like this can be developed according to the risk profile. Hint: high-risk flaws should be resolved first.

There is, of course, some real value penetration testing has which stems from probing a system in its final operating environment. Uncovering environment and configuration problems (and concerns) is the best result of any penetration test.

So, should you conduct penetration testing? Absolutely. It's an important and necessary security activity. But any kind of "penetrate and patch" mentality is insufficient as a standalone software security approach. It is much more powerful in tandem with training—partially based on penetration testing results—design review, code review, and security testing at the integration level. A well-structured software security initiative does all of those things and uses penetration testing to demonstrate that all those other things generated the expected level of quality.

Myth 4: Software Security Is a Cryptography Problem

Software security isn't the same thing as security software. You can use a crypto library to add a security feature to an application, but that's not the same thing as making an application secure. The liberal application of magic crypto fairy dust to your code will provide no security by magic (in fact, the same myth applies to any particular security feature, not just crypto). Software security needs to be built in from the ground up. As such, cryptography makes the list of software security myths at number four.

Cryptography can be a useful tool when it comes to securing data, communications, and code globs (among other things), but it's no silver bullet. Why, you ask?

Magic crypto fairy dust

The idea that you can sprinkle magic crypto fairy dust all over a piece of software and it will then be secure is wrong. First off, security is a system property, not a thing. Simply adding crypto measures to your code is unlikely to make it secure. Additionally, cryptography is astonishingly complex to get right. Not only is the math difficult, applied cryptography is filled with sneaky pitfalls that are easy to get wrong.

Cryptography can't find or eradicate bugs and flaws, but sometimes it can temporarily obscure them and make life that much more difficult for debuggers and architects. Crypto can't train your developers. And crypto even falls prey to penetration testing from time to time.

To give you an example: if a SQL injection is found in your app that talks to an encrypted database, do you think encrypted data or plain text data will be returned?

Just as we've already discussed with perimeter security, security tools, and penetration testing, crypto alone is not the answer. (Or more generally, any particular security feature alone is not the answer.)

There are two reasons for this.

1. Most attacks against applications do not target security features specifically (and crypto is just another security feature). Instead, most target defects in other parts of the application (bugs and flaws) in any part of the reachable code. Using crypto does nothing to fix these defects or to protect them from exploit. All SSL does is protect an exploit from prying eyes as it whizzes by. It might even allow an attack to sneak right by some firewalls.

2. Applied crypto is designed to protect data, not the application that processes data. Data in motion between machines can be protected. Data at rest on a disk can be protected. But data being manipulated in real time by a running application is a sitting duck, or a running duck—well, you get the picture.

> Data being manipulated in real time by a running application is a sitting duck.

Is security a feature or a function? No.

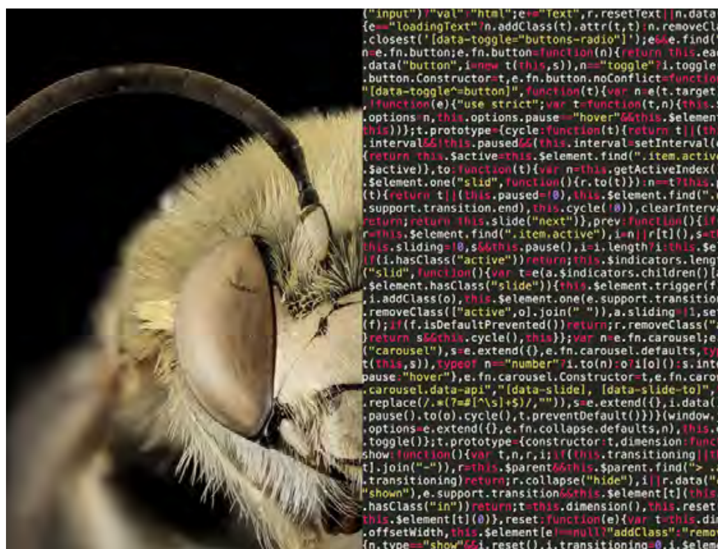
Developers and software architects have been trained for years to piece out their work in terms of features and functions and often think by default that security is one of the two. The most common security feature in a developer's mind is cryptography. That is, when you ask a developer to make their code secure, often the first thing they think of is crypto.

Software security is about integrating security practices into the way you build software, not simply integrating security features into your code. The integration of identity management features, multi-factor authentication or PCI compliance measures are all very useful and meaningful tools that shouldn't be discounted. But at the end of the day, simply adding security properties such as these won't magically secure the software.

There is no single 'fix' that guarantees security. Achieving the most secure software requires carefully administered, step-by-step measures that take place strategically throughout the SDLC.

MYTH 5

Myth 5: It's All About Finding Bugs In Your Code



Are bugs a big deal in the software security conversation? Absolutely. Implementation bugs in code account for at least half of the overall software security problem. Finding and fixing those bugs is an essential software security initiative (SSI) practice. This brings us to the 5th myth of software security.

The 50/50 divide

If bugs are only half the problem, what's the other half entail? The other half involves a different kind of software defect that occurs at the design level: flaws.

© 2017 Synopsys, Inc.

For a great set of flaws and how to avoid them, see the work of the IEEE Center for Secure Design.

The division of design flaws and bugs is about 50/50. Both need to be secured to ensure your software's well-being. You can institute the best code review program on the planet, with the strongest tools known to humanity, but it's unlikely that you will be able to find and fix flaws this way. As the domain of threat modeling and architecture analysis, flaws must be found and resolved by experienced people. These activities have resisted practical automation so far in the software security evolution. As a new and continuously evolving field, software security measures have historically been identified as security needs and trends arose.

Finding a problem is different than fixing a problem

While finding bugs in your code is great, and there are variety of methods for finding bugs (including penetration testing and the use of security tools), unless you fix what you find, security doesn't improve. Finding issues is only the beginning. Sadly, it's incredibly common for firms to find issues without taking any measures to fix those issues. Part of the problem is that technology developed to find bugs often lacks helpful remediation recommendations for fixing the issues. For example, static analysis tools find thousands of different bugs, but in no way fix them.

Broaden your horizons. Find the bugs—not limiting your search to only 10—and most importantly fix the bugs! It's also important to make sure you train your developers not to make new bugs every day; otherwise, you're setting up a hamster wheel of pain. And remember, bugs are only half of the equation. Make sure to also focus attention on flaws occurring at the design level.

Where do I start?

Want to know what other organizations are up to with regards to security? The BSIMM was designed to measure and understand real world software security initiatives. Quantifying the practices of organizations throughout a variety of industries, we can describe the common ground shared by many as well as the variations that make each unique.

Learn how to get involved in the BSIMM study.

Myth #6: Security Should Be Solved By Developers

Who should 'do' software security?

- Security people?
- Compliance people?
- Developers?
- None of the above?
- All of the above?

Sadly, solving software security is not as easy as picking only one population to hang the security problem on. In fact, software security does (eventually) take "all of the above," but ultimately must be coordinated by a central software security group (SSG).

We know from the Building Security In Maturity Model (BSIMM) that a software security initiative (SSI) should be led by a SSG. In fact, a SSG is a necessity to execute software security, application security, product security, or whatever you'd like to call it. The point is, creating a SSG is the first step when your firm is ready to tackle software security.

Strategizing your SSG

The notion that developers—and only developers—should collectively and magically be responsible for software security sounds great in theory, but it never actually works in practice. So if a SSG is not just a bunch of developers, what does it look like?

Here are a few key things to consider when forming your SSG:

- What kinds of people should join to conduct necessary security functions?
- How big should the SSG be?
- How do we even organize it in the first place?

We asked real firms how they went about organizing their SSG. At the fall 2014 BSIMM Community Conference in Monterey, California, we posed these questions to the BSIMM Community. During a break-out session, complete with plenty of cocktails to get the conversation going, 23 firms presented and discussed their SSG structure. Once we heard from all presenting BSIMM firms, five major groups emerged:

Services

Seven of the presenting firms built a SSG to provide software security services. This is the most common organizational approach in our data set. These SSGs are structured in terms of specific services they provide to the rest of the organization (i.e. penetration testing, code review, architectural risk analysis). Among the seven firms in this category, the most commonly provided service was code review with static analysis.

Policy

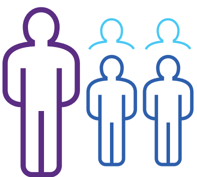
Five firms built a SSG around setting policy. This approach sets policy that is to be enforced by technologists outside of the SSG. For example, the SSG may declare that code review is mandatory and must be completed with no high-risk vulnerabilities resulting, but does little to steer development groups directly in how to get this done. In many cases, the SSG doesn't hold much power beyond what they can influence, and they take a "set a goal and hope" approach.

Business Unit

Two firms built a SSG mirroring business unit organizations. This approach is a more difficult way to get things up and running. The idea is to provide the same kind of help to each of the business units in an organization, tailored to their way of working.

Hybrid

Four groups built a SSG with a hybrid policy and services approach. In some cases, the SSG finds itself conflicted in a sense that it sets policy that is mandatory, but doesn't have the service bandwidth to execute the work for all the teams that want it. This leads to questions like: "who pays for what?" and "who gets the blame when delays occur because of lack of software security service bandwidth?"



Management

Three firms with very mature initiatives had SSGs structured around managing a distributed network of others doing software security work.

And just for the record, the two remaining firms (among our 23 total presenting firms) were organized in unique ways found only in their firms. In some sense, replicating this approach is not really possible.

A SSG for the long term

Now that you know the first step is to form a SSG, the next step to take into consideration is to make sure the SSG includes both software people with deep development know-how, security people with strong architectural knowledge and people who can interact with the business.

It should be obvious from what we've said here that the idea of simply training all of the developers is another one of those "yes, do that, but not only that" software security best practices. Should you train your developers and arm them with software security know-how? Yes! But you also need to check code for bugs, check design for flaws and check systems for security holes. Why? Well, aside from the obvious reasons, your developers don't write all the code you deploy. Even if your code is perfect, all that vendor code and other stuff created in groups without a SSI will trip you up every time.

MYTH 7

Myth #7: Only High-Risk Applications Need to Be Secured

Our seventh and last myth of software security is about scale. Today's application portfolios are often quite large—thousands of apps. Getting started back in the day meant identifying those apps that carried the most risk and focusing all of the attention on them. However, those days are over.

Today it's about securing the entire portfolio—the overall attack surface. No matter whether you're working on a small project or a huge corporate application strategy, the key to reasonable risk management is to identify and keep track of risks over time as a software project unfolds.

Risk management is the mature way to think about information security due to the fact that it's a careful balance between security and business functionality. Too much security can ruin a good business proposition by bogging it down in a security swamp. Conversely, not enough security can lead to unacceptable levels of risk which can threaten the business itself.

Like any executive, a CISO's role is to make risk management decisions and then communicate those decisions to the executive team and the company. These decisions should be in alignment with the corporate culture, compliance and regulatory mandates, and the community standards of acceptable diligence. Not to mention that they must also be technically sound.

Unfortunately, risk management can fail, and when it does, it really fails! Let's discuss failure conditions involved with risk management and how to avoid them in the future.

Failure 0: An exposed portfolio

Security controls are kind of like clothing for software, shielding it from the harsh elements. Imagine that you have a large portfolio of "naked" software assets to protect. As most of us do, you have a limited budget to spend on clothes. Thus, the risk management problem is one of determining how many, and which type, of clothes to pair with your software assets.

One option is to divide the portfolio into categories of risk. Three simple and effective buckets are: high, medium, and low. This division can be accomplished by first tagging the software assets with categories such as: Internet-facing, directly-controls PII, handles money, life threatening, schedules conference rooms, etc. Once the portfolio's assets have been categorized, add them to one of the three buckets.

The question then becomes what to do about imposing security controls on each of the buckets. Let's take testing as an example. In this case, controls can include things like: architecture risk analysis, code review, dynamic analysis, and penetration testing. You decide to allocate scarce resources such that you apply all controls possible to the "H" bucket, a handful to the "M" bucket, and you ignore the "L" bucket completely.

The types of risk management decisions made for these buckets make all the difference between risk management reasonableness and failure. In Figure 1, you'll notice that only the "H" bucket assets have security controls imposed, and those are imposed linearly according to the perceived risk inside the high bucket. The problem is that no "M" or "L" assets are considered for controls. This is a dangerous scenario.

If most of your portfolio falls below the "do nothing" line (in Figure 1 this would be the x-axis), you have a problem. A simple way to resolve this is to implement a few easily-applied security controls, across the board into all software assets. In other words, none of your assets should be naked. At least give them minimal security controls—equivalent to security underwear.

From Figure 1, you'll want to move forward by moving the security control minimum line "up" and we want to flatten the curve by bending the low end up as well. That will result in something along the lines of Figure 2.

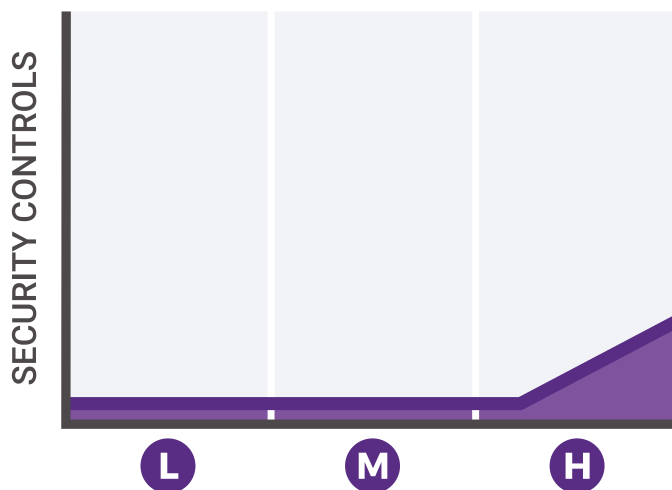


Figure 1

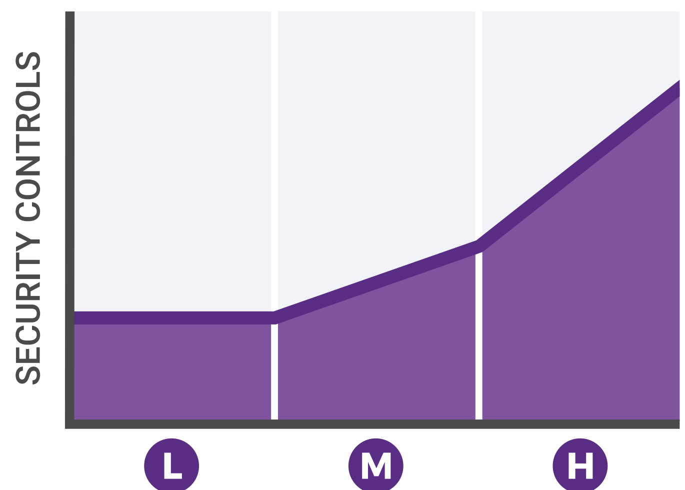


Figure 2

© 2017 Synopsys, Inc. | www.synopsys.com |
SYNOPSYS®

Failure 1: How many medium risks make high risk software?

There is a bug severity problem in software. How many little bugs does it take to make a big bug? Sometimes, too many little bugs make code so sketchy, nobody wants to get near it. No individual bug is a show-stopper by itself, but together they add up to create the perfect storm. And this problem is nothing new.

The problem of compounding influences security risk and risk management decisions. How many low or medium risk software assets can you stand? What's the reasoning for classifying something as "medium" risk? Does risk compound like bugs seem to? Can your portfolio be as sketchy as bug-infested code?

Consider your software asset risk curve. If your buckets are properly configured, your assets will likely fall into a standard distribution and end up resembling the Bell curve in Figure 3. On the other hand, your Bell curve may resemble Figure 4 if your categorization scheme isn't optimal.

If the nature of risk for your organization is unique, the number of medium risk assets calls for a uniform set of security controls to be put in place. Few high risk assets means that a high-only strategy makes little sense.

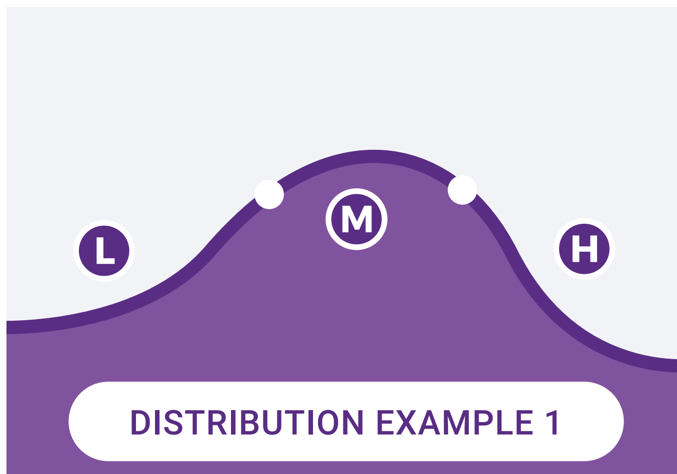


Figure 3

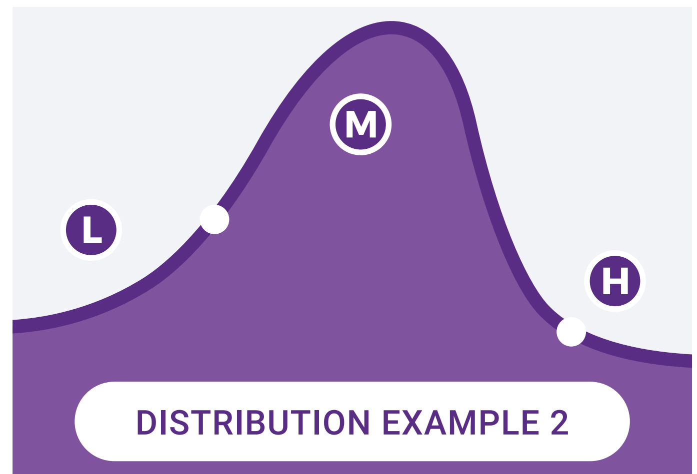


Figure 4

Here's the plan

Ultimately, both risk management failure conditions discussed here can be addressed by the same guiding principles:

- leave no code naked
- tighten security controls between low and high risk assets to the extent possible without moving the high risks down the prioritization list
- create a specific testing schedule (even if it's once every couple of years for internal low-risk applications).

The key that any organization must understand is that they need to cover their entire software portfolio—both the high-risk and the low-risk apps must be taken into consideration.

SYNOPSYS®

The Synopsys difference

Synopsys helps development teams build secure, high-quality software, minimizing risks while maximizing speed and productivity. Synopsys, a recognized leader in application security, provides static analysis, software composition analysis, and dynamic analysis solutions that enable teams to quickly find and fix vulnerabilities and defects in proprietary code, open source components, and application behavior. With a combination of industry-leading tools, services, and expertise, only Synopsys helps organizations optimize security and quality in DevSecOps and throughout the software development life cycle.

For more information, go to www.synopsys.com/software.

Synopsys, Inc.
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

Contact us:
U.S. Sales: 800.873.8193
International Sales: +1 415.321.5237
Email: sig-info@synopsys.com

©2020 Synopsys, Inc. All rights reserved. Synopsys is a trademark of Synopsys, Inc. in the United States and other countries. A list of Synopsys trademarks is available at www.synopsys.com/copyright.html. All other names mentioned herein are trademarks or registered trademarks of their respective owners. October 2020