The Rugged Implementation Guide







Strawman Edition

August 2012

Acknowledgements

Rugged was created by Josh Corman, David Rice, and Jeff Williams in 2010.

Join the Rugged Project!

This guideline is a strawman version intended to generate discussion, comment, controversy, and argument. We expect significant changes to occur in future versions. The Rugged Project is actively seeking anyone interested in finding new and better ways to create secure software. Please join us and contribute your ideas.

http://www.ruggedsoftware.com

Acknowledgements

We gratefully acknowledge the support from Aspect Security, who drafted the initial version of the Rugged Implementation Guide.

License

The Rugged Implementation Guide

This guide is intended to help you and your organization reliably produce Rugged software. You should, prior to reading this document, review "The Rugged Handbook," which details the security challenges with modern software development and describes a new way of thinking, or culture about achieving security.

The purpose of this implementation guide is to provide some practical guidance in creating the security story defined in the Rugged Handbook, show how application security best practices fit together into a coherent program, and provide ideas for optimizing your program in the future.

There are lots of "best practices," "security requirements", and other gems in this document. Some of these are innovative, but most were invented elsewhere. Most of these ideas have not been presented in the context of an entire program, linked all the way back to the business reason for doing them. Putting these ideas, like "should we digitally sign our code?" in context. Otherwise everything in security becomes an emergency "must-do" requirement.

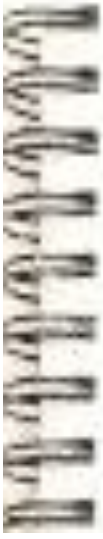
Scope

The primary focus (for now) is on individual projects doing new development of applications. Rugged can very easily be used for a variety of different types of organizations and projects, but we have to limit the scope and perform trials before we are willing to expand the scope.

In the future, we imagine expanding the scope of the Rugged approach beyond just software applications to IT infrastructure. We also need to make it work for legacy software projects. In the future, we will expand the scope of the project so that it applies more naturally at the enterprise-level, across a portfolio of custom-code applications and products, instead of just individual projects.

Overview

In this guideline, we are going to work through the process of developing a security story for RuggedBank, a financial organization building an Internet facing application that stores personal information, such as name, address, email, account numbers, and financial data.

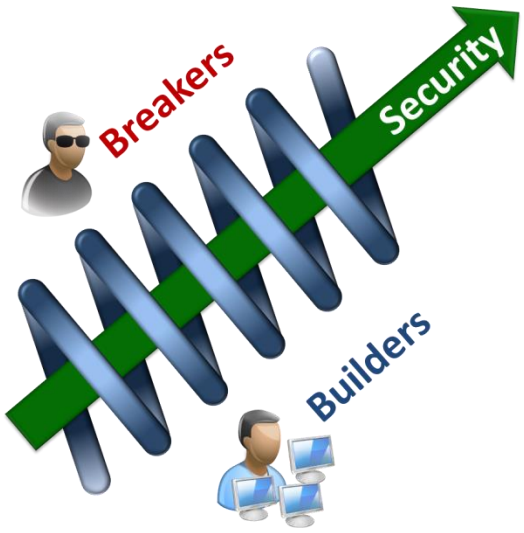


Example pages from RuggedBank's security story will be shown with a spiral notebook motif to make it clear that they are only one example. We encourage experimentation with security story formats and appreciate your feedback on what is effective and/or ineffective.

Rugged Culture

Rugged software development is intended to evolve your security as quickly as possible. The key to creating a culture that can cause this fast evolution is to enable "builders" and "breakers" to both collaborate and compete with each other in a friendly and cooperative way. We believe that this approach will lead to better and more cost-effective security.

Builders are the engineers that design and implement security defenses. Breakers, on the other hand, are engineers that find ways to bypass defenses that could cause damage to the business. Builders and breakers are just roles, and a single engineer might switch back and forth between them.



There is no one Rugged organizational structure. Instead, the goal is to set up an ecosystem that creates a virtuous feedback loop of breaking, building, breaking, and building. This ecosystem will generate strong security as a byproduct.

Getting builders and breakers to collaborate and compete with each other on security without animosity or blame is a challenge. We believe the right approach is to make security explicit and visible to the entire team. The discussion of security should not be allowed to become personal. Instead, the discussion should always be framed in the context of a security story.

For example, if a SQL injection hole is discovered in production, the response should not be blame, but an investigation into what could be changed in the security story so that problem would be prevented and detected much earlier in the software lifecycle. Every security problem discovered affords an opportunity for improvement.

Many organizations have a morass of standards, policies, defenses, and technologies that make it impossible to know what "secure" actually means. The Rugged organization knows exactly what security means because they have a concrete story explaining it.

The security story doesn't have to be terribly lengthy or complex. Choosing the right strategies can keep security simple. For example, many organizations struggle with SQL injection flaws. They implement complex input validation, web application firewalls, intrusion detection systems, integrity checking, and more to try to stop it. But using parameterized queries everywhere is a simple strategy that is easy to implement and verify. Strategies like this keep your security story simple, short, and effective.

The Story

The story idea consists of four layers that will help you structure and organize your security information. The model provides a line-of-sight from the things you care about to everything you do in your application security program. Remember, security only works if everyone is aligned, and the only way to align people is to clearly express your story.

1. **Lifelines**. First, we will work out RuggedBank's "lifelines." These are the cornerstones of the development organization and information technology that are critical to protecting the business, such as ensuring that sensitive data is not disclosed, the system remains available for users, and that development staff are trustworthy. The rest of the story is driven by these business concerns.
2. **Strategies**. The next step will be to capture a "strategy" for ensuring each of those lifelines are firmly in place. The strategy is a model or approach for ensuring that a concern is protected. For example, one strategy for protecting sensitive data would be to use encryption everywhere. Another might be the choice not to store sensitive information. Still another would be to rely on strong authentication and access control. The strategy should also explain why the concerns are sufficiently protected against the expected threats. Strong strategies feature defense-in-depth, simplicity, verifiability, etc… The choice of strategy is critical to making the rest of the process cost-effective.

3. **Defenses**. Once we have strategies defined, we need to choose specific "defenses" that implement each strategy. Together the list of defenses details how the strategy will actually be implemented. This should include the specifics of technologies, design patterns, and other implementation details. For example, if the approach is to encrypt, the defenses should list exactly the encryption product, algorithm, key size, key management, and other aspects of the actual defense.
4. **Assurance**. Finally, the "assurance" is specific proof that the defense is in place, correct, and used everywhere necessary. The assurance evidence is generated as part of the SDLC and during operations, and the story should link to specific evidence that demonstrates each defense is implemented correctly. Examples of appropriate evidence include output from SAST, DAST, or IAST tools, penetration test and code review reports, threat models, architecture review reports, third party analysis, or custom tool output.

Of course you don't have to do these steps in this strict top-down order. The goal is to get something down quickly and then immediately start iterating and improving the story. You start by capturing the best security story possible from the information currently available. Sometimes this initial story has major obvious gaps, missing defenses, unimplemented strategies, and other serious problems. Many different roles collaborate to create this security story. Executives and business owners share lifelines, security analysts contribute threat and vulnerability analysis, architects design and position defenses, developers implement controls, and testers verify they work properly.

Stating Your Commitment

Your story should start with the "why" behind security. Maybe you care about security because you had a major breach, or maybe because you are afraid of a compliance violation. Hopefully, you can identify some reasons that will be more inspiring to your staff and customers. Think about what's important in your industry. Is it honesty? Quality? Customer service? Innovation? Or something else?

Try to capture "why" security is important to you first. It's important that clients, employees, customers, executives, and everyone else understand the importance of the story. Keep asking why until you arrive at some fundamental truths about your company, your industry, and your customers. It's okay to summarize your primary lifelines so that people get what you're talking about, but don't spend too much time on "what" your security measures are or "how" you will achieve it.

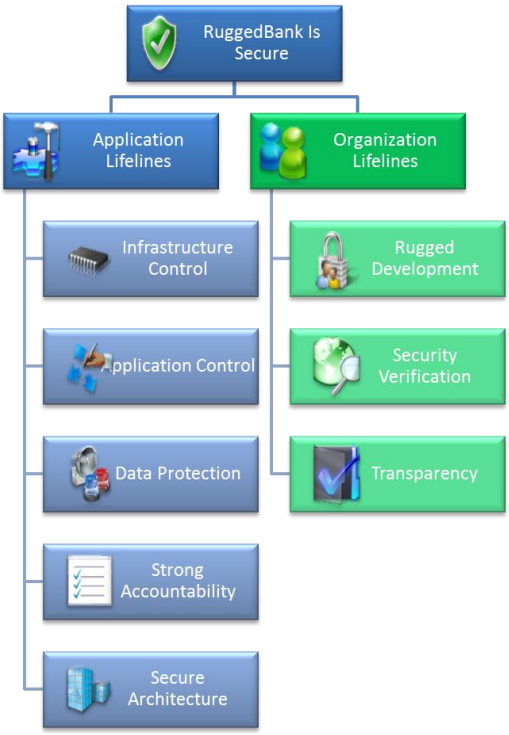
Our Commitment to Security

Brick-and-mortar banks protect money with vaults, guards, and alarms. RuggedBank envisions a future of online financial transactions that provides both better security and more power to consumers. RuggedBank is committed to the highest standards of application and network security for our RBOnline application. We secure our infrastructure, ensure data is always protected, build defenses against injection and other application attacks, practice rugged software development, and carefully verify our code. At the core of our security is a commitment to transparency – across our protections, processes, and even potential problems.

Identifying Your Lifelines

The first step in creating a security story is to identify your lifelines. Think of your lifelines as a set of statements that, if true, would ensure that you and the other stakeholders in your application are secure. There are two types of these lifelines:

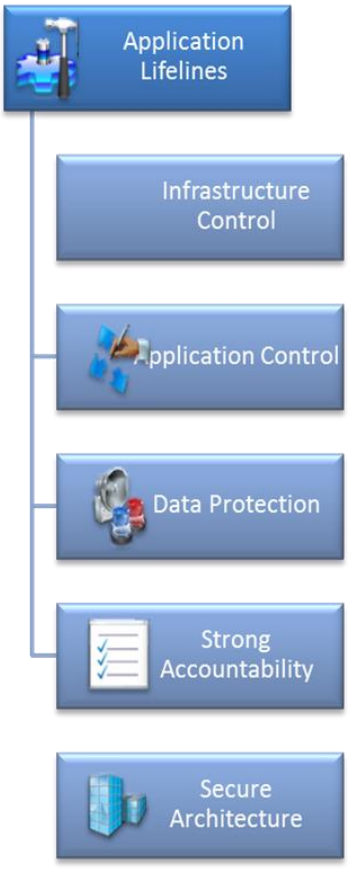
- **Application Lifelines** – An information technology lifeline focuses on the technical protections that directly support the interests of the stakeholders in the application. These lifelines are typically associated with security defenses for confidentiality, integrity, and availability.
- **Organization Lifelines** – A development organization lifeline focuses on attributes of the organization(s) that create, maintain, and operate the application. These are supporting lifelines in that help to ensure that all the other lifelines are strong, rather than directly protecting the interests of the stakeholders themselves.



We recommend starting with the structure to the right, which you can refine as you work through the discussion below and evolve your security story over time. On the left, you can see the primary high level defenses RuggedBank is relying on. On the right, you can see security relevant aspects of the organization that is creating and operating the software.

In the next few pages, we will explore how to identify your lifelines.

Identifying Application Lifelines



There are a variety of techniques for identifying security threats, attackers, threat agents, vulnerabilities, assets, capabilities, and other important details. Unfortunately, they are all over the map with regard to the level of abstraction, and none of them is particularly well suited for figuring out a good set of high level priorities might be. We'll use a hybrid approach that uses the best of threat modeling, abuse/misuse cases, architecture review, and security engineering to reverse engineer a set of business security concerns.

For the moment, let's remember that your security story is very likely to grow and change over time, so it's not important that you get it perfect on the first attempt. What is important is that you write down a best effort and then gather extensive feedback to make it better. You should target a handful of high level business concerns and later stages will flesh them out. For many applications, your results are likely to look like the Application Lifelines shown in the diagram to the right. If you're building a nuclear reactor or a medical device, you will probably want to change this up a bit.

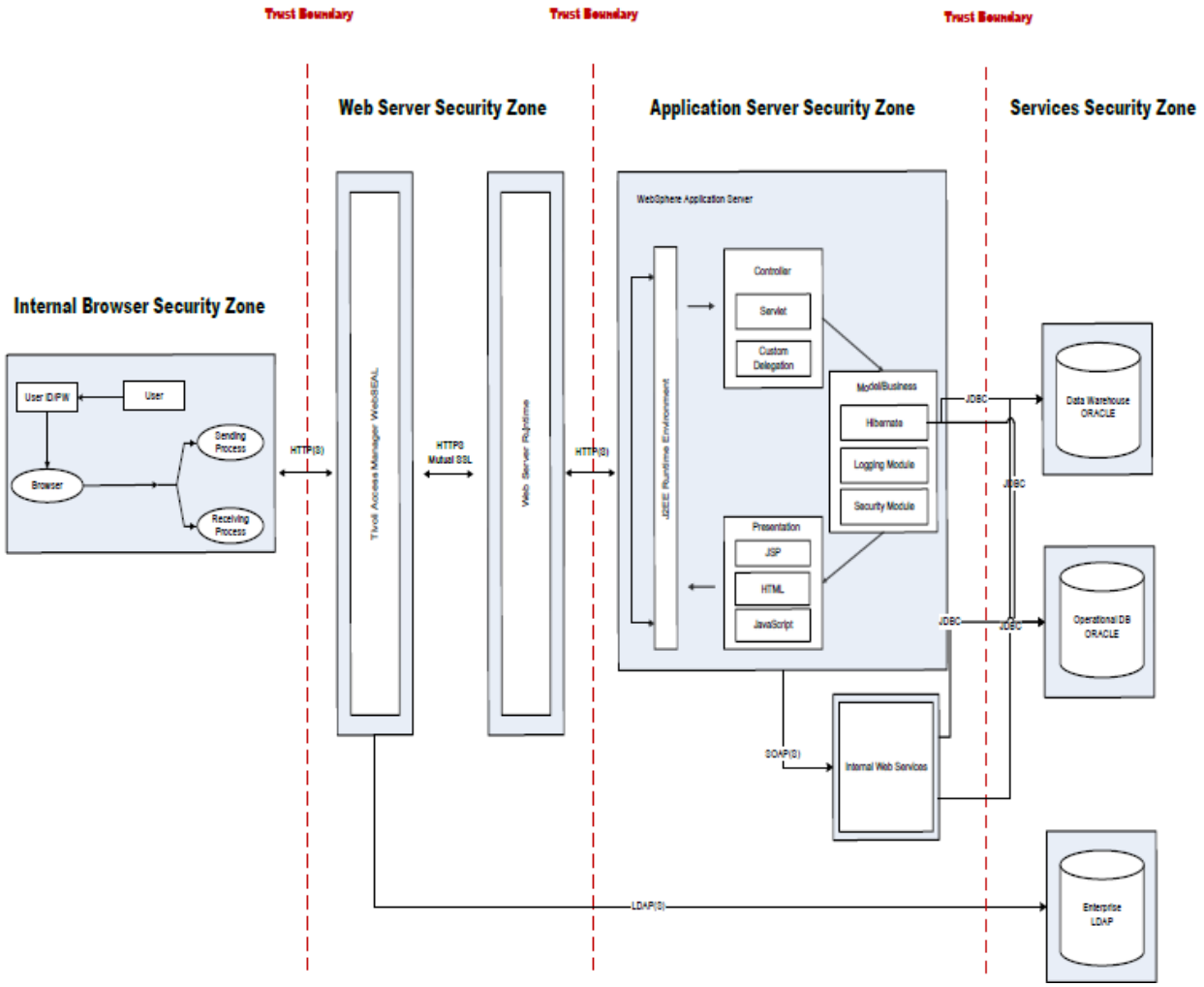
Step 1: Create a list of stakeholders

First, it's always useful to consider all stakeholders of your application's security. The most obvious is your own business. But don't forget about the direct users of your application, people whose information is processed by your system, shareholders, owners of connected systems, people that rely on the information and services provided, auditors, and others that might rely on the system.

For each of stakeholder, consider the system from their perspective. What would they consider a disaster if it happened? What could possibly happen to the system that would cause them harm? In addition to financial loss, don't forget to consider the loss of intellectual property, reputation damage, legal and compliance exposure, and the harm to morale that might occur if an attacker was successful.

Step 2: Make a high level architecture diagram

Your goal is a diagram that looks similar to the one below. You're going to use it as a brainstorming tool, so you really only need the high-level components here – don't get too detailed. You want to identify all the components and the *possible* connections between them. Remember that if there's nothing *preventing* a connection to a server, then anyone can connect to it. You should show each of the hosts, and then the major pieces of software on each computer. If the system exists, then you might be able to get an existing diagram to use, although in most cases it will be better to create your own. If the system does not exist, then sketch out a hypothetical architecture even if it is just one big box.



Step 3: Add threat agents

A risk happens when a threat agent can adversely affect one of your assets or capabilities. So we're going to seek paths through your software architecture that could connect a threat and something you value. Start by considering all the possible hostile threat agents in your environment. Don't forget your "authorized" users, as they might be malicious insiders or their environment might be compromised.

If your system already exists, consider your current defenses – why are they there? Did you forget some threats? Good chance there's a reason that will lead you to an important business concern. Are you logging? Why? Perhaps you are interested in accountability for legal reasons. Are you using LDAP to manage entitlements? How many roles does your application have? Have you considered all the layers and components in your architecture?

Step 4: Add data and capabilities

Next, figure out where sensitive assets are located in your software architecture. Seek out data stores, file servers, optical data farms, log files, message queues, and anywhere else that data might come to rest, even temporary stores and caches. Trace all the places through the system where the data flows,

and understand which connections handle sensitive data as well. You should also identify any functions or capabilities that you consider critical. For each asset, consider exactly what it is that you want to protect. Many times it is the confidentiality of information that is the primary focus. Address the integrity and availability of your data and capabilities as well. You may also want to consider accountability as an important property.

Step 5: Consider intangible assets

Perhaps what's most important to your organization is protecting their reputation as a careful steward of other people's information, and ending up in the Wall Street Journal would be a disaster. Or perhaps, for your organization, compliance with legal or regulatory requirements is the primary concern. Retaining control of your infrastructure and application are key assets that need to be protected as well. Brainstorm all the terrible things that could happen in the worst case scenario. Forget about what protections you already have in place, we'll be adding those back in later. For now, just focus on what you want to protect.

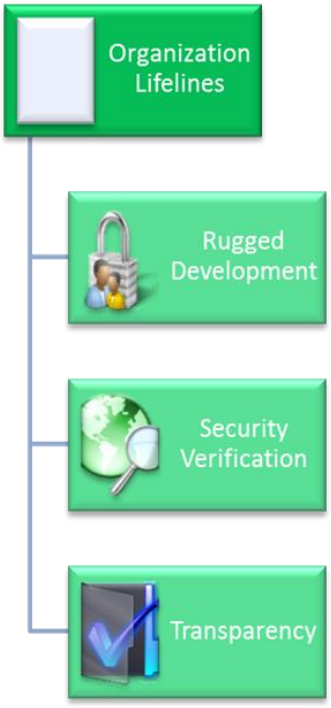
Step 5: Rip, mix, and burn a balanced list

You want to end up with a handful of top level technology lifelines. Five would be manageable, ten less so. Some areas can be combined if the top level is too lengthy or confusing. Or other areas might need to be split in order to balance the importance of the different lifelines. You want to structure them so that if they are all clearly satisfied, then you will be satisfied with the security posture of your application.

Step 6: Rinse and Repeat

Of course, identifying security concerns is not actually a linear process. You'll want to work top-to-bottom and bottom-to-top repeatedly until you are sure that you have all of your primary concerns covered by these lifelines. While you may work out the bulk of your security story in the first cut, there may be updates and even major refactoring as new threats emerge and application architectures are modified. For example, if you take an internal-only application and decide to make it internet-facing, it's time for a hard look to ensure all the necessary lifelines are in place.

Identifying Organization Lifelines



Your organizational lifelines describe the security-relevant aspects of the people, process, and technologies (or tools) that comprise your software development organization. You're trying to explain why your organization is set up to reliably produce secure software. While the technology lifelines can be different for each application, the organization lifelines are generally similar. This doesn't imply that every development organization is the same, just that the top level lifelines are similar.

It's important to remember that this entire set of lifelines provides support to direct lifelines focused on the defenses in the application itself. At best, this section can only show that the organization does the right things, hires the right people, or uses the right technologies. Nevertheless, this assurance should be quite compelling. Many people would feel more secure with a strong organization and unknown software than the other way around.

Rugged Development

In this section, you should outline how security works in your software development organization. What is the organizational structure and how are responsibilities assigned? Who does security architecture, verification, and management? How are people hired, trained, background checked, and kept up-to-date? Is there a code of ethics that people are required to sign? What tools are used to write, manage, build, test, security test, and integrate the software? Where does your security intelligence come from? Ideally, a Rugged development organization will be designed not only to produce secure code, but to evolve security over time.

Security Verification

We chose to break out security verification separately since it plays such a critical role in Rugged organizations. In this section, you should describe how your verification organization works, what metrics are tracked, how vulnerabilities are managed, what tools and techniques are used, and how independence is maintained. The best stories will describe how the verification team works together with the development team, yet maintains a competitive atmosphere to drive evolution.

Transparency

We have found, over many years, that although it might seem prudent to keep the details surrounding security secret, this ultimately undermines security efforts. Unfortunately, keeping security secret does not help solve them. Conversely, when security is practiced in the sunshine, everyone gets a chance to evaluate and improve the system. At a minimum, making the details of security available internally in your security story will help everyone on the development team communicate and collaborate easily.

Making security details public, on the other hand, causes concern in many organizations. While there is not much sense in protecting information that attackers can easily figure out, certain assets, such as source code may contain valuable intellectual property. Although applications should be secure even if attackers are in possession of the source code, it seems prudent for many organizations not to release their code. In this section, you should affirm your commitment to transparency and the basic elements of your program.

The Overview Page

Most of the people interested in the security of your application are only looking to see that you care about what matters to them. They want to see that you have identified the right lifelines and have a strategy in place for dealing with them. They may drill down into an area or two, but for the most part they are only interested in the high level story. We recommend a one-page highly organized summary of the rest of your security story.

The Location

The security story should be located an obvious place for the intended audience (internal or external). We recommend that every business locate their main security story at "/security" in their main website, for example http://www.example.com/security. If the business operates multiple applications, then each should have their own security story that possibly extends the main business story – http://www.example.com/app/security.

The Commitment

The page starts with a high level statement articulating the organization's commitment to application security and the high level strategy. We recommend really thinking about *why* security is actually important to your clients and your sector, not about your business in particular. You want to share your high-level strategy for achieving this commitment with a few sentences about your overall approach. Keep this paragraph short – the details are to follow.

The Lifelines

Below the commitment, include a high-level description of the information technology and development organization lifelines you have established. Along with each area, a brief summary of the approach is recommended. Writing up each of the lifelines is fairly easy, the trick is figuring out exactly what the right lifelines are for a particular application.

The Contact

Below the commitment we recommend establishing a central email address to capture and handle questions or comments. This is also a good place to let people know how to report security vulnerabilities.

Example Overview

On the next page is an example of an overview page. This is not the only way to format or organize this information. Remember that there are many different stakeholders in your security story, and you want a starting page that can meet all of their needs.

The Rugged Bank Security Story

***Commitment**: Brick-and-mortar banks protect money with vaults, guards, and alarms. As we move these operations to cyberspace, everyone should expect even better protection of their money and financial information. RuggedBank is committed to the highest standards of application and network security for our RBOnline application. We secure our infrastructure, ensure data is always protected, build defenses against injection and other application attacks, practice rugged software development, and carefully verify our code. At the core of our security is a commitment to transparency – across our protections, processes, and even potential problems.*

Infrastructure Control

RBOnline security depends on maintaining control of our physical and network infrastructure. We are hosted in a secure data center, managed by trusted staff, and our systems are kept hardened, patched, and up-to-date. Our network is defended and segmented by firewalls and we detect and block both network and application attacks.

Learn more... ▸

Application Control

All our data protections depends on having software that is resistant to injection or other attacks that rob us of control of our systems. We have strict rules around data handling and interpreter use to prevent injection. In addition we ensure the application is always available and ready for use.

Learn more ... ▸

Data Protection

Protecting your data is our highest priority. Our primary defense is universal authentication and access control. As a secondary defense, we also use strong encryption everywhere data is transmitted or stored. To ensure that these protections are not bypassed, we have established extensive protections against injection and other attacks.

Learn more ... ▸

Rugged Development

To make sure that our application is designed and implemented securely, We follow a secure development lifecycle. All our developers are trained in using our standard security defenses. We have performed extensive threat modeling and minimized our attack surface. We use powerful tools to protect and manage our source code, user stories, and software artifacts.

Learn more ... ▸

Security Verification

An independent team performs architecture analysis, code review, and penetration testing on our application. We use a custom test suite and automated security tools to double-check for vulnerabilities. We are committed to finding and eliminating vulnerabilities throughout our secure development lifecycle.

Learn more ... ▸

Transparent Security

Without information, good security decisions are impossible, so we are committed to ensure that you understand the protections that we have provided. We will notify you of any security issues that might affect your business. You can export your data from our application at any time, and we will purge it from our systems.

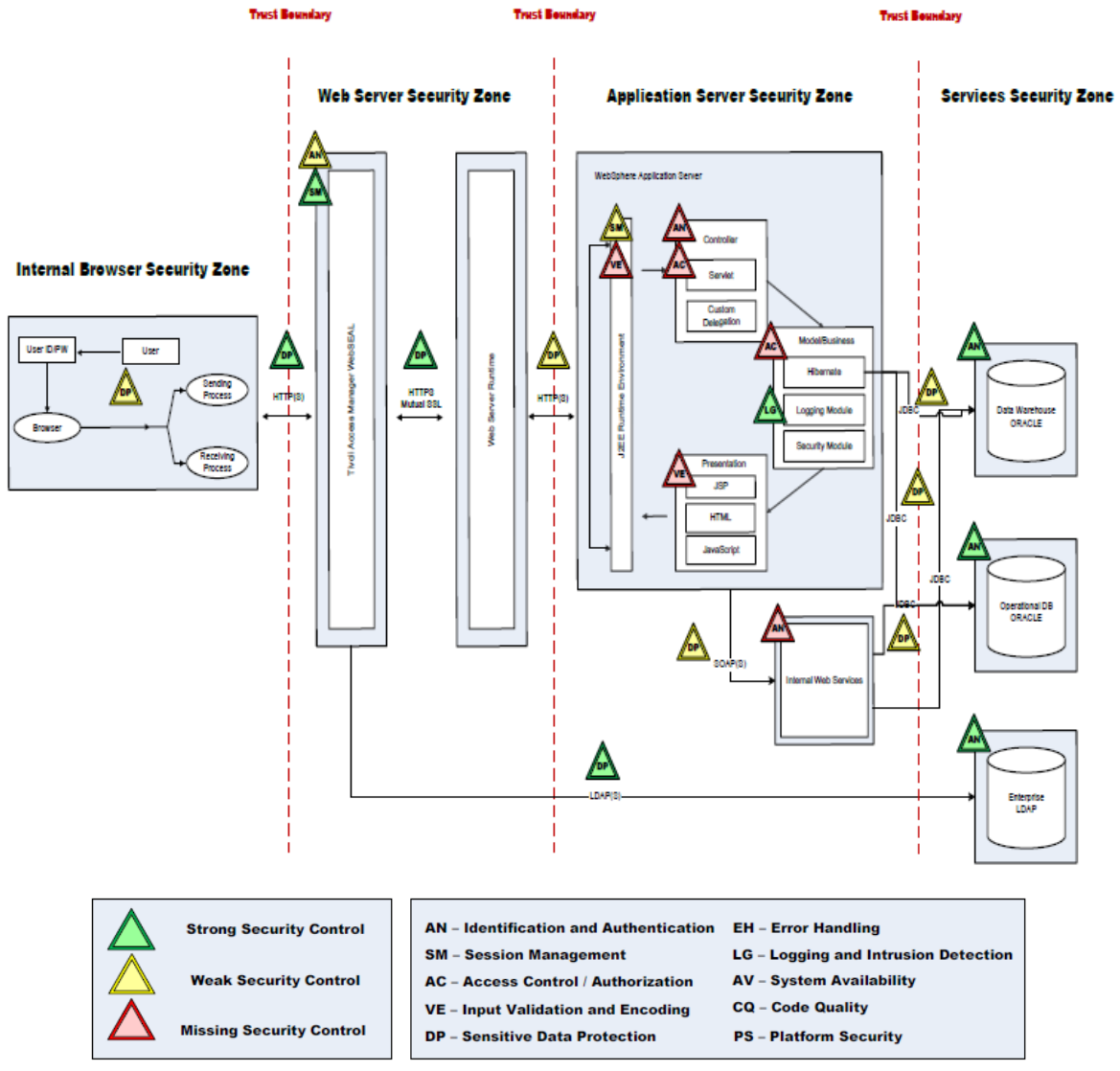
Learn more ... ▸

If you have questions, comments, or want to report a security vulnerability, please send email to security@ruggedbank.com.

Creating a Security Strategy

Once you've identified the lifelines, the next step is to create a security strategy for ensuring that each lifeline is strong and complete. The goal is to articulate a clear and compelling description of how the lifeline will be implemented.

How would you protect a castle against unwanted visitors? Your strategy might involve a moat, guards, archers, boiling oil, etc… Your defenses have to work together to provide effective protection. The strategy is an approach for how it all works together to completely cover the lifeline. Strong strategies feature defense-in-depth, simplicity, verifiability, etc… Your choice of strategy is critical to making the rest of your story cost-effective. You may want to build on an architecture diagram like the one below.



Step 1: Choose primary defenses

Your primary defenses are the countermeasures (e.g. infrastructure, services, libraries, custom code) that squarely address the threats to this lifeline. For example, your primary defenses for protecting data might be minimizing data collection and access control. Together, these two defenses ensure that only authorized people will be able to access sensitive data.

Step 2: Determine secondary defenses

Your secondary defenses should provide protection even if the primary defenses fail. This is often referred to as "defense in depth," but is rarely practiced. For example, everyone knows that we should protect passwords when they are stored. However, this is really a secondary defense. The primary defenses should prevent attackers from accessing the password file in the first place. Secondary defenses might be simply defense-in-depth where they provide an alternative or weaker protection than the primary defense. Or secondary defenses might be detective or deterrent mechanisms that discourage an attacker from attempting an attack or help to minimize the damage caused in the event of a successful attack. For example, intrusion detection is a secondary detective control that enables quick response before serious damage can occur. While relying entirely on intrusion detection would be foolhardy, it is an extremely useful secondary defense.

Step 3: Ensure that known vulnerabilities are addressed

In order for these defenses to fully defend a lifeline, we have to ensure that they are implemented correctly and in all appropriate places. Your story should include a discussion of why these defenses should be trusted. Ideally, you will use strong, proven defenses, and your story can simply reference that fact. If you are creating new defenses, you should provide the details about how they are designed and built.

Step 4: Keep it simple and verifiable

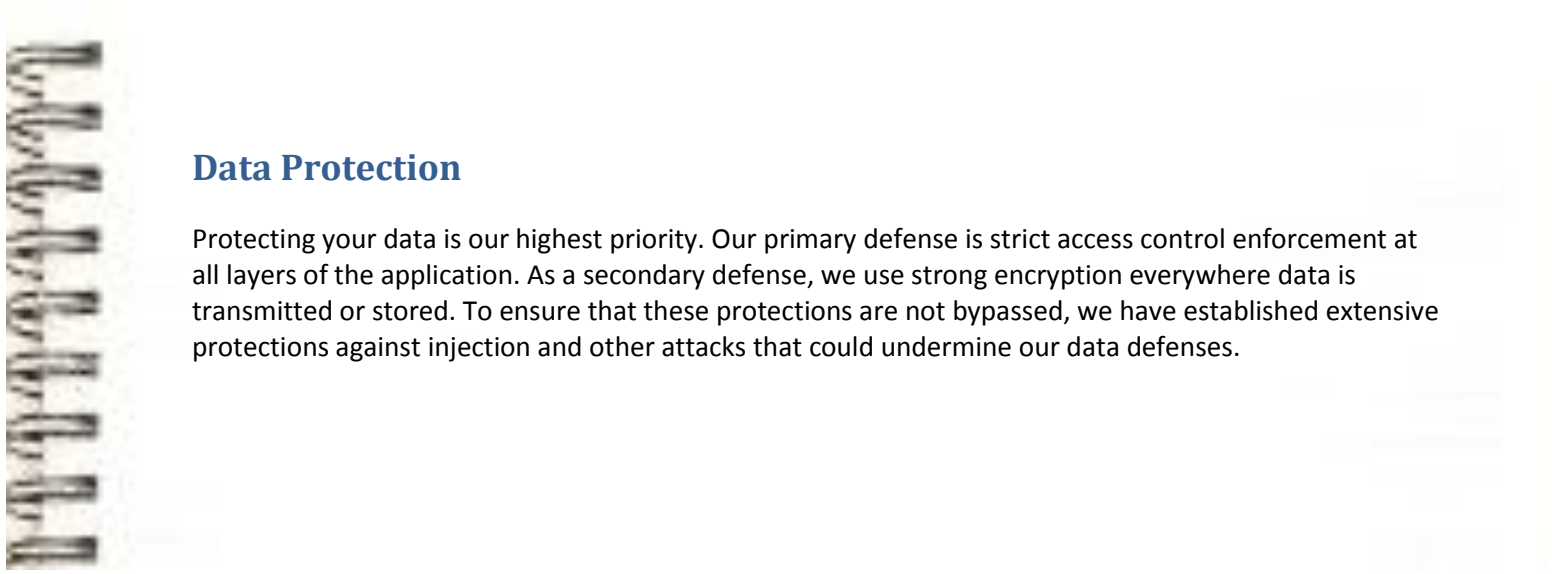
You should be prepared to fight to keep your strategy simple. Complexity is the enemy of security. For example, choose to use authenticated SSL everywhere, rather than trying to pick and choose which connections will be secure.

Step 5: Challenge aggressively and constantly

Technology and Security are incredibly fast moving industries. Even while you work to secure existing technology, new, non-Rugged, technologies introduce new risks to be mitigated. To stay abreast of these security challenges, you must ruthlessly challenge your own security story. You analyze the security story for threats you didn't consider, were not considered by the developers, integrators, or anyone else in the chain, defenses you should have put in place, opportunities to establish defense in depth, weaknesses that you didn't avoid, tests you should have performed, and better ways to automatically gather evidence.

Example Strategy

The following is an example of a strategy as part of a security story. Although it is relatively short, this story conveys considerable information about how data protection will work within the application and how it will operate in the organization.



Data Protection

Protecting your data is our highest priority. Our primary defense is strict access control enforcement at all layers of the application. As a secondary defense, we use strong encryption everywhere data is transmitted or stored. To ensure that these protections are not bypassed, we have established extensive protections against injection and other attacks that could undermine our data defenses.

Articulating a Security Defense

A security strategy might involve a number of specific defenses. A defense is sometimes referred to as a countermeasure, a security mechanism, or even control. It may help to use an adjective to qualify the defense, such as "Strict Access Control," "Universal Authentication," or "Trusted Staff."

Step 1: Craft a "mini-strategy"

To help the reader understand the purpose of this defense and how its pieces fit together, it helps to provide a few sentence overview of the purpose and structure of the defense. The goal is that everyone should understand *why* this defense is necessary and how it works.

Step 2: Identify the technologies involved

For each defense, you should capture exactly what defenses you are using and why they are the right choice for the application. For example, instead of a vague requirement to "do role-based authentication," your story should indicate that the application uses Oracle Identity Manager Version 2.2.1 and any supporting technologies. You should say how the defense is integrated and how it is used. For example, the application uses a custom JavaEE filter that calls OIM to control access to every function.

Step 3: Describe how the defense is supported and maintained

Understanding who created and maintains the defense will help readers understand organizational responsibility for security defenses. If a security bug is found or implementation advice is needed, people need additional information about where to go and who to ask. For example, many organizations might use a standard cryptographic library to hash passwords before storage. But developers may need advice about whether to use bcrypt or SHA-1 or some other algorithm, how many iterations to use, and where to store the salt.

Step 4: Describe how the defense is configured and managed

Security defenses often require configuration and system administration. Indicate any critical settings that are required for the operation of this defense. Even the best defenses can be undermined by careless or incorrect configuration. These configurations need to be carefully examined and managed as a part of the configuration of the system, so that they do not "degrade" over time. Discuss what organizational entities will actually perform the configuration and management of the defense.

Example Defense

The following is an example of an access control defense as part of a security story.

Strict Access Control

We use a positive access control model across all the layers of RBOnline. This means that users may only access web links, business functions, data, and user interface elements that they are explicitly authorized to access.

1) An access control matrix showing which roles are allowed to access which web links, functions, data, and user interface elements is maintained by the Business Access Management team.
2) All access entitlements are stored in Oracle Identity Manager (OIM) 2.2.1 and are managed by our Business Access Management team. Entitlements are granted only when necessary for job functions, for a limited time, and reviewed upon any change in employment status.
3) Access to all web links is controlled by SiteMinder 10, which gets information from OIM and allows access only if the user is specifically authorized for that web link.
4) All programmatic access to functions, data, or user interface elements is mediated by our custom AccessControl module which is maintained by the AccessControl team. The AccessControl module gets information from OIM and allows access only if the user is specifically authorized.
5) To prevent attempts to bypass access controls, direct object references such as filenames and database keys are not exposed to end users. Instead, indirect references that cannot be abused are provided.
6) Secure hosting facility personnel have limited access to protected data in an encrypted form for support and backup purposes.

Assurance Strategy and Results

A key part of your story is evidence that it represents reality and is not just a paper exercise. So you should figure out how you are going to demonstrate to yourself, your organization, and others that what you say in your story is true. There are a range of options for verifying a defense. At one end, there is complete mathematical proof using a formal model of the security defenses. This is the approach taken in the Orange Book at levels B2 and above. The other end of the scale is the most trivial proof that a system is secure, such as an assertion from the developer that they used a secure approach.

The good news is that many assurance strategies are quick and easy. Want to prove that your application is not susceptible to SQL injection? Run a simple tool that demonstrates that all queries are performed with a parameterized interface and bind variables are used. You get fairly high assurance for a minimal cost. Build this custom scan into your continuous integration environment and you'll always have up to the minute proof of your assurance. This type of approach can work for many different defenses.

Step 1: Think positive, then negative

When verifying a security defense, the first step is to make sure that the defense is present, correct, and properly applied. After which, it makes sense to start looking for common flaws and other negative testing approaches. Perhaps it *seems* easier to run a negative scanning tool to look for problems. But it's not. There are thousands of ways that applications can do security wrong, and these tools need to understand them all. But if you define secure behavior in your story, verification only has to establish that *one* positive behavior.

Step 2: Ensure defenses are always invoked, tamperproof, and easy to verify

At a minimum, verification should establish that the defense is actually present in the application. This test would fail, for example, if the application requires encryption and there is no cryptographic module in the software. While trivial, many applications fail this simple test.

Better verification will establish that the defense mechanism is always invoked, tamperproof, and verifiable. Proving that an access control is always invoked might involve comparing the intended access control matrix to one generated from the software (either by testing or static analysis). Showing that it is tamperproof might be accomplished by extensive testing and fuzzing. And establishing that it is verifiable should show that the design is relatively simple and understandable.

You'll also want to demonstrate resistance to all the known attacks and vulnerabilities in similar defenses. For example, an input validation method should properly canonicalize the data to prevent tunneling of encoded attacks.

Finally, the verification should ensure that the defense is used properly in all the places where it is needed. For example, an output encoding defense should be used in all the places where untrusted output is sent to an interpreter (like the browser).

Step 3: Gather both direct and supporting evidence

There are two kinds of evidence that can be used to support a security story. Direct evidence comes from the code itself, and includes test results (like unit tests, functional tests, and regression tests that cover security defenses), quality assurance test case results, static analysis results, quality measures, and the like.

Supporting evidence, on the other hand, comes from the software development organization and its processes rather than the code itself. Supporting evidence is critical as both defense in depth and as support for areas of the story where direct evidence is difficult or expensive to generate. This evidence may include business processes, architecture diagrams, data flow diagrams, and their associated explanation detailing how a specific security goal is achieved.

Both direct and indirect evidence can be generated by a third-party. This might be through the use of a tool or service, such as an independent security assessment (DAST, SAST, IAST, security code review, or penetration test) or something provided by a supply chain provider. This might also be the output from a set of test cases built by another group. While third party evidence can be very powerful, it's important to remember that the story is your responsibility.

Third party attestations by an expert are also possible. In this case, the credentials, expertise, and impartiality of the expert ought to be shared. More importantly, the strength of the attestation depends in part on disclosing what the expert actually did. For example, did the expert review the code? Look at architecture diagrams? Talk with a developer? Or simply get paid to attest to the security of the product sight unseen. Beware "seals" and "certifications" that don't provide these details as there are services on the market that offer such attestation for money with arbitrary analysis to support them.

Making Assurance Efficient

You will want to consider the most cost-effective ways to prove that the defense works. As your security story develops you should improve. Remember, verification doesn't have to be perfect. Sometimes a penetration test is the best solution possible within the time and schedule. By capturing the verification strategy in the story, we make it clear to everyone what is going to be checked and how.

Often, your security story will depend on someone else's security story, such as a component vendor, hosting provider, or other supply chain providers. One example is when you use a product created by another organization (perhaps an open-source group) and you do not have any specific proof that the product is secure. This applies to the entire supply chain. In this case, you can test the product yourself, ask the provider for a security story, or decide that you will accept the risk associated with trusting a component without any particular reason to do so.

Over time, you can seek to improve the automation and quality of the verification checks. Ideally, we reach a state where all the security defenses are automatically verified across the lifecycle.

Example Assurance

The following example shows how the results of running a tool provide verification of a defense in your story. In this case, a simple tool scans to ensure untrusted data is protected by an input validation method. Verification results are not always going to be perfect, at least at first, but over time the team will improve the verification levels. Note, that in this example, we are building directly on the Application Control defense, and linking to an external source which contains the output of a custom tool that performs an automatic review of input validation code. This tool could be a simple "grep" like tool, or something more sophisticated like a dynamic scanner, a static analysis tool with a custom rule, or a runtime monitoring tool.

Application Control

1) We use ESAPI canonicalization and input validation on all untrusted data regardless of the source, then create data objects from the data instead of using Strings. View automatically generated input scan…

Security verification tool output (example tool output linked to above)

Method	Total	Validated	Vulnerable
HttpRequest.getParameter()	455	405	40
HttpRequest.getHeader()	25	25	2
HttpRequest.getCookie()	6	6	1
…			

Appendix A: Templates, Examples, and Ideas….

In this section, we have drafted some examples for use as a starting point for crafting your own security story, so this whole section uses the spiral notebook motif. You should revisit the how-to sections above for guidance on how to think through your own application project and come up with a compelling argument.

Technical Lifeline

Strategy

Infrastructure Control

Our security depends on maintaining control of our physical and network infrastructure. We are hosted in a secure data center, managed by trusted staff, and our systems are kept hardened, patched, and up-to-date. Our network is defended and segmented by firewalls and we detect and block both network and application attacks.

Primary Defense

"Mini-Strategy"

Physical Security

We protect the physical security of RBOnline because an attacker who gains physical access could undermine our other defenses. Therefore, we require authentication, restrict access, and use a variety of security devices to ensure accountability and detect intrusions.

1) RBOnline is completely hosted at the [DATACENTER].
2) Physical access to systems is restricted to authorized personnel.
3) [DATACENTER] follows generally accepted industry best practices including: restricted access, mantraps, alarm systems, interior and exterior cameras.
4) The cameras and alarms for each of these areas are centrally monitored for suspicious activity
5) Facilities are routinely patrolled by security guards.

You can verify our physical security by evaluating our datacenter's security story for yourself. We also maintain a log of all datacenter events that are noteworthy to our business.

Summary Verification

Primary Defense

Hardened Network

Host and network security is a critical foundation for the security of our applications and data. We keep our technology up-to-date and hardened.

1) All commercial software and operating systems are hardened and all patches and updates are applied regularly. Review all product versions.
2) All servers are protected by [SOLUTION] for both realtime and interval virus scanning.
3) Strong passwords are used to manage servers
4) Passwords are encrypted in both storage and transmission
5) Infrastructure is protected by both network and application firewalls configured to allow only authorized traffic and block application attacks

Add verification evidence inline

6) Internally, the network is segmented to isolate each set of systems
7) We use dedicated intrusion detection devices to provide an additional layer of protection against unauthorized system access.

Intrusion Detection and Response

There is always the possibility that an attacker will discover a vulnerability that we never considered in the technologies on which we rely or even in our own code. To help mitigate the damage that this might cause, we also use an industry regarded intrusion detection system and monitor our systems for attack.

1) Our application identifies attacks in progress by examining security events in realtime, such as input validation errors, access control errors, and other exceptions that a normal user would not typically generate.
2) We monitor the activity on our network, hosts, and applications using [SOLUTION] IDS using an up-to-date ruleset.
3) When issues are reported, our network security group investigates and responds appropriately.
4) Authenticated users that attack our application are blocked from further access until the incident can be investigated.
5) In the unlikely event that we find evidence of a breach, we are committed to providing users and other stakeholders with the full and timely information about what happened and how we responded.

Application Control

Preventing injection ensures the integrity and control of our servers. We have strict rules around data handling and interpreter use to prevent injection. In addition we ensure the application is always available and ready for use.

Strong Input Validaton

Validation limits the attack surface of our application. Although we do allow special characters when required by the business, our validation greatly restricts the range of possible attacks. In addition, we rely on input validation to detect attacks in progress.

1) We use ESAPI canonicalization on all data as part of input validation.
2) We consider all data from any source, including internal sources, untrusted until it has been through a rigorous validation process.
3) All input is validated against a strict specification of what should be allowed using the ESAPI validator architecture.
4) After validation, we create data objects from the data instead of using Strings.

Injection Defenses

Injection attacks could enable attackers to bypass our access controls and encryption defenses. To prevent injection, we use parameterized interfaces where possible and escaping where it is not. In all cases, we perform strict input validation and are careful configuring parsers.

1) To prevent SQL injection, we use Hibernate for our database storage, and have been careful to avoid patterns that might allow injection within Hibernate. View our secure coding standard.
2) Our XML parsers are carefully configured to avoid attempting to resolve entities inserted by attackers.
3) Our web interface is primarily HTML with JQuery. Data is fetched from the server using Ajax and the data is inserted into the DOM using safe methods.
4) We have enabled Content-Security Policy for browsers that support it, and do not use inline Javascript

Always Available

We have taken extensive steps to ensure that the application will remain available, even in the face of excessive load and typical denial of service attacks.

1) From a network perspective, we defend against denial of service attacks and balance the load across a set of servers.
2) We have automatic failover between datacenters in two separate parts of the United States.

3) Our strict authentication of all traffic limits denial of service attacks to registered users that can be quickly identified and blocked if necessary.

4) All unauthenticated traffic is static or has been optimized to incur minimal load on our servers.

Data Protection

Protecting *your data* is our highest priority. Our primary defense is universal authentication and access control. As a secondary defense, we also use strong encryption everywhere data is transmitted or stored. To ensure that these protections are not bypassed, we have established extensive protections against injection and other attacks.

Minimal Data Collection

Minimizing the amount of data collected keeps our attack surface small and limits the damage in case of a breach.

1) We only collect the data absolutely necessary, specifically [LIST OF DATA COLLECTED AND HANDLING DETAILS]
2) Credit cards are collected and stored in RAM only during the pendency of a transaction. Once approved, all credit card information is purged from our systems.
3) Data is centralized and stored only in our database. We do not have any other persistent data stores, including file systems, caches, or message queues.

Strict Access Control

We use a positive access control model across all the layers of [APPLICATION]. Users may only access web links, business functions, data, and user interface elements that they are explicitly authorized to access.

1) An access control matrix showing which roles are allowed to access which web links, functions, data, and user interface elements is maintained by the [Business Access Management] team.
2) All access entitlements are stored in Oracle Identity Manager 2.2.1 and are managed by our [Business Access Management] team. Entitlements are granted only when necessary for job functions, for a limited time, and reviewed upon any change in employment status.
3) All operations designated “critical” require two people to be involved
4) Access to all web links is controlled by SiteMinder 10, which gets information from OIM and allows access only if the user is specifically authorized for that web link.
5) All programmatic access to functions, data, or user interface elements is mediated by our custom AccessControl module which is maintained by the AccessControl team. The AccessControl module gets information from OIM and allows access only if the user is specifically authorized.
6) To prevent attempts to bypass access controls, direct object references such as filenames and database keys are not exposed to end users. Instead, indirect references that cannot be abused are provided.

7) Secure hosting facility personnel have limited access to protected data in an encrypted form for support and backup purposes.

Universal Authentication

We believe that everything that happens on our systems should be fully authenticated and traceable to an individual. We ensure that all credentials, tokens, and identifiers remain within an encrypted and authenticated "bubble" that extends from our server to the client. This "bubble" is comprised of defenses that do not allow credentials to be forged, spoofed, leaked, guessed, misused, or otherwise compromised. Even if a session credential is leaked, we require full re-authentication for sensitive functions and will detect anomalous behavior and prevent access.

1) All identity management is performed within Oracle Identity Manager 2.2.1 by our [Business Access Management] team, including password recovery, two-factor authentication, account lockout, and logoff functions. Entitlements are granted only when necessary for job functions, for a limited time, and reviewed upon any change in employment status.
2) All LDAP queries are executed over an authenticated LDAP/S channel using strong encryption. View security test results.
3) We have password strength checks and failed login lockouts to ensure that RBOnline is not susceptible to brute force attacks. View security test results.
4) Passwords are salted and hashed using bcrypt to prevent exposure in the unlikely case that our credential database is compromised.
5) We use per-session random tokens to prevent request forgery. View security test results.
6) We require re-authentication for sensitive functions, to minimize damage from compromised accounts. View security test results.
7) We use the default session identifier cookie for our application servers and protect it with the secure and http-only flags. View security test results.
8) We rotate session identifier cookies upon successful login
9) We protect our user interface from being framed by including a framebreaking script as well as using the X-FRAME-OPTIONS header on every page.

Encrypt Everything in Storage and Transit

Our primary defenses minimize collection and control access, but we also use strong encryption to ensure that all of the data we store is inaccessible to attackers. Our model is to encrypt all data whether in storage or transmission and to carefully protect the encryption credentials.

1) All data is stored on dedicated disk volumes that use TrueCrypt for full disk encryption. We extend the use of encryption to backups, logs, and any other data. View disk status report.
2) All client connections use strong encryption and mutual authentication, to protect against sniffing, spoofing, and other communications attacks. View SSL Labs results.

3) All backend connections are also both encrypted and mutually authenticated, to protect against insider attacks and attackers using compromised internal hosts. View network scan results.

4) Internal certificates are managed by a dedicated certificate authority.

Strong Accountability

We have two goals in the accountability area. First, we want to ensure that every single transaction occurring within our systems is strongly associated with an authenticated identity. This includes both customer and employee actions within our systems. Second, we want our logs to be comprehensive enough to identify and diagnose attacks upon our system.

Log Capture and Review

We use a combination of human and automated review to analyze our logs to detect attacks.

1) All logs are centralized in LogLogic SIEM.
2) Our logs capture a full record of activity on the system.
3) Logs have a 3 day retention policy on each server.
4) Log alerts go to our SOC immediately for analysis.
5) Manual log review takes place on a weekly basis.

Accountability Logging

Our accountability logs include a full record of every action taken by users upon our system. We log enough details of each transaction so that we have a full record of what has transpired.

1) Both failed and successful operations are logged, unless the amount of log traffic generated provides only negligible benefit.
2) Nightly backups of logs and data are moved to another dedicated encrypted volume and are retained there for 14 days. After 14 days they are retained on our secure encrypted backup service for 3 years.

Security Logging

Our security logs ensure that if we are attacked, we will capture enough information to enable us to detect attacks and diagnose what damage was caused.

1) We log all input that could be used to cause an attack.
2) Each log event includes the time, associated IP addresses, identity information, responsible code module, and exceptions.
3) We automatically ratchet up log capture when attacks are detected.

Secure Software Architecture

Our architecture uses the latest versions of proven software components that have been hardened according to the latest guidance. Our custom code leverages standard security mechanisms that are either part of the platform or are provided by our centralized security API.

Hardened Software Stack

Our software architecture uses Java 7, Tomcat 7, Spring 3.1.1, Spring Security 4.1, Log4j, and mysql 8. View entire technology stack. Java provides us a variety of security benefits, including automatic garbage collection, structured exception handling, and strong typing.

1) We harden these technologies by disabling all unnecessary services and configuring all available security controls to their most secure setting.
2) We ensure the integrity of our secure configuration by checking integrity hashes and deploying our entire stack automatically with Puppet.
3) We keep all of our components up-to-date by carefully monitoring the mailing-lists and websites for each of these technologies.
4) We also monitor for vulnerabilities filed against these technologies to ensure that we are not exposed by new discoveries.
5) To minimize the effectiveness of malicious code, we run with the Java SecurityManager enabled and have a policy that restricts libraries from doing things that are outside their intended purpose.

Standard Security Defenses

By externalizing our security defenses, we simplify the writing of secure code for our developers. Where possible, we have integrated these defenses into our framework, so the security happens automatically for developers. In other cases, we have established a common pattern for invoking the standard defenses at the appropriate point. As described below, our defenses have been extensively tested and verified to ensure that they are correct.

1) Our security API extends the widely used OWASP Enterprise Security API (ESAPI) and includes the following defenses: authentication and session support, access control, input validation, output escaping, logging, encryption, and many other defense mechanisms. These defenses protect against a broad range of application security vulnerabilities, including the OWASP Top 10, SANS Top 25, and others.

Rugged Software Development

Our software development organization is designed to make sure that our application is designed and implemented securely. In our culture, everyone is responsible for creating code that is consistent with our security story. We also constantly challenge the security story and make improvements to simplify and strengthen it. We constantly measure our software against our security story and treat discrepancies as bugs.

Organization Structure

Everyone on our development team is responsible to understand our security story and perform their tasks accordingly. In addition, we also have a Software Assurance Manager and a Software Assurance Engineer assigned to our project with primary responsibility for planning and maintaining the security story. We rely on the enterprise security architecture team for advice and guidance on selecting and implementing security defenses. We also rely on the enterprise security testing team for independent reviews of the software we produce and guidance on remediation.

[INCLUDE SECURITY ORG CHART HERE]

Trusted Staff

We are careful about the people involved with building, maintaining and operating [APPLICATION], as they will have a certain degree of access to the code and operational system. However, we minimize that access wherever possible, such as not allowing developers access to production systems.

1) New employees must interview with at least four existing employees and come with excellent recommendations from previous employers.
2) We verify education, previous employment, and references
3) We conduct criminal, credit, immigration, and security checks
4) Must sign our Software Assurance Code of Ethics
5) Every project has a Software Assurance Manager and at least one Software Assurance Engineer

Improving the Security Story

In our development team, there is not a strict delineation between “builders” and “breakers.” In fact, some engineers regularly are involved with both finding weaknesses in the story and designing security defenses to improve security.

1) We identify concerns, architect defense models, implement defenses, and verify security.
2) All new and modified security-relevant code is reviewed by the project Software Assurance Engineer.
3) All of this is captured in our Security Story (this document).

Security Training

Our entire team, including management, development, and operation have a deep understanding of the security required for [APPLICATION].

1) All of our developers have been trained in the full range of application security vulnerabilities and the best approaches for controlling them, including the OWASP Top 10, OWASP ASVS, and MITRE CWE.
2) All of our managers, architects, and operations staff have mandatory application security training appropriate to their role.
3) All staff are required to take role appropriate eLearning modules and pass a certification test.
4) All of our employees are regularly trained to protect the confidentiality and privacy of customer information, and required to execute an agreement to that effect.
5) We keep up to date on the state of application security by attending conferences, participating in open source communities, and doing our own security research. View recent projects.

Development and Build Tools

Our development, build, and deployment environment is critical to the security of [APPLICATION]. We use our tools to help us ensure integrity from source to operation. We also strive to detect flaws as early as possible in the development process.

1) Subversion – fully authenticated. All code, configuration, documentation, and all other artifacts are stored in configuration management.
2) Eclipse IDE
3) Maven build definition
4) Sonar repository
5) Jenkins continuous integration server
6) Quality and security analysis nightly (Findbugs, PMD, Checkstyle)

Security Verification

An independent team performs our architecture analysis, code review, penetration testing processes on our application. We use a custom test suite and automated security tools to double-check for vulnerabilities. We are committed to finding and eliminating vulnerabilities from our application throughout our secure development lifecycle.

Verifiable Design

1) All of our security controls are designed to be easily verifiable. They are as small and well-structured as possible to facilitate manual and automated analysis.
2) Our security controls are all data-driven, so that the configuration data can be managed separately from the defense itself.
3) All security controls are capable of exporting their current configuration and enforcement status for the purpose of verifying against the design.
4) All of our security controls are managed separately from our business applications. Controls are well defined and integrated into our software frameworks to the maximum extent possible.

Security Architecture Review

A security architecture review of our application is performed annually to identify existing flaws and to evaluate how new threats might affect the security of our application.

1) An independent team of Zoobar consultants periodically performs a detailed applications security architecture review of the full environment with the goal of identifying any security weaknesses and design simple, strong defenses.

Automated Security Tests

During development, we run an extensive custom test suite and a variety of dynamic and static security tools to verify the security of [APPLICATION]. Our continuous integration process ensures that this verification is performed along with every build, and alerts go to the development team immediately whenever a security problem is identified.

1) Our primary defense is an extensive set of Selenium test cases that exercise every security control to verify that it is present, correct, and used everywhere necessary. View latest report.
2) To ensure our controls will ensure security even in unexpected conditions we have created a “Simian Army” to aggressively tamper with our systems in random unexpected ways and verifying that no security exposures result.
3) As a double-check, we use Barfoo static analysis engine in our development environment. We have over 100 customized rules to recognize our security controls. View latest report.

4) Our final check is to run Foobar dynamic scanning product on both test and production servers to detect security mistakes early in development as well as our actual deployed environment. We have tailored the scan to our application and achieve 64% coverage. View latest report.

Manual Verification

Our primary defenses are strong technical and non-technical security defenses. But we are strong advocates of careful checking to ensure that these defenses are present, correct, and properly used in all the right locations.

1) Developers are responsible for security testing and delivering code without vulnerabilities. We provide all developers with OWASP Zap Proxy and expect them to verify that their security defenses work. View report of vulnerabilities discovered by lifecycle phase.
2) To help expand the coverage of our verification efforts, Zoobar Consulting performs both network and application penetration testing and code review on our application on a periodic basis. We schedule a review before major deployments and upon any major change to a security critical component. View latest report.

Tracking and Remediation

We carefully monitor risks to ensure they are tracked from inception to closure. We want to make all of our software as resilient to threats as possible. So when we discover a security problem in one part of our code, we work to eliminate it everywhere.

1) We track all of our security issues in Archer. View recent reports.
2) We stamp out all instances by writing new test cases, tuning our tools, enhancing our frameworks, and adding awareness training for developers. View recent vulnerabilities.

Transparent Security

Without information, good security decisions are impossible, so we are committed to ensure that you understand the protections that we have provided. We will notify you immediately of any security issues that might affect your business.

Sharing Security Details

We believe in sharing the details of our security measures. We are careful not provide attackers with information they couldn't figure out for themselves, and these details might help you evaluate our service against the alternatives. We have found tremendous benefit in capturing our security story and using it as a way to align our efforts to secure [APPLICATION]. You'll find that everyone in our organization has a strong appreciation for security and everyone is involved in ensuring that [APPLICATION] stays protected. Sharing these details with you is just the logical extension of our internal practices.

Bug Bounty Program

In order to help ensure that our defenses are thoroughly evaluated by the best in the business, we have created a bug bounty program that enables talented security researchers to explore a test version of our application and find security holes. This provides us with useful information about vulnerabilities and rewards the researchers for their hard work.

[See Google, PayPal, Mozilla for examples.]

Security Communication

We pledge to keep you up to date about any security issues that we might encounter in any part of RBOnline. You have the right to evaluate each issue and make the decision about whether it affects your business or not. Organizations with healthy security organizations do identify vulnerabilities occasionally in their products. We also believe that security communication should work in both directions. Please don't hesitate to contact us with your ideas about improving our security, report vulnerabilities, or just ask questions. Please email us at security@ruggedbank.com.