



black hat[®]
USA 2018

AUGUST 4-9, 2018
MANDALAY BAY / LAS VEGAS



 #BHUSA / @BLACKHATEVENTS

SDL That Won't Break the Bank

Steve Lipner

Executive Director

SAFECode

Lipner@SAFECode.org

- Introduction
- SDL history and origins
- SDL big picture
- SDL for smaller/newer organizations
- Basic Principles
- What to do next
- Some final basics
- Conclusions

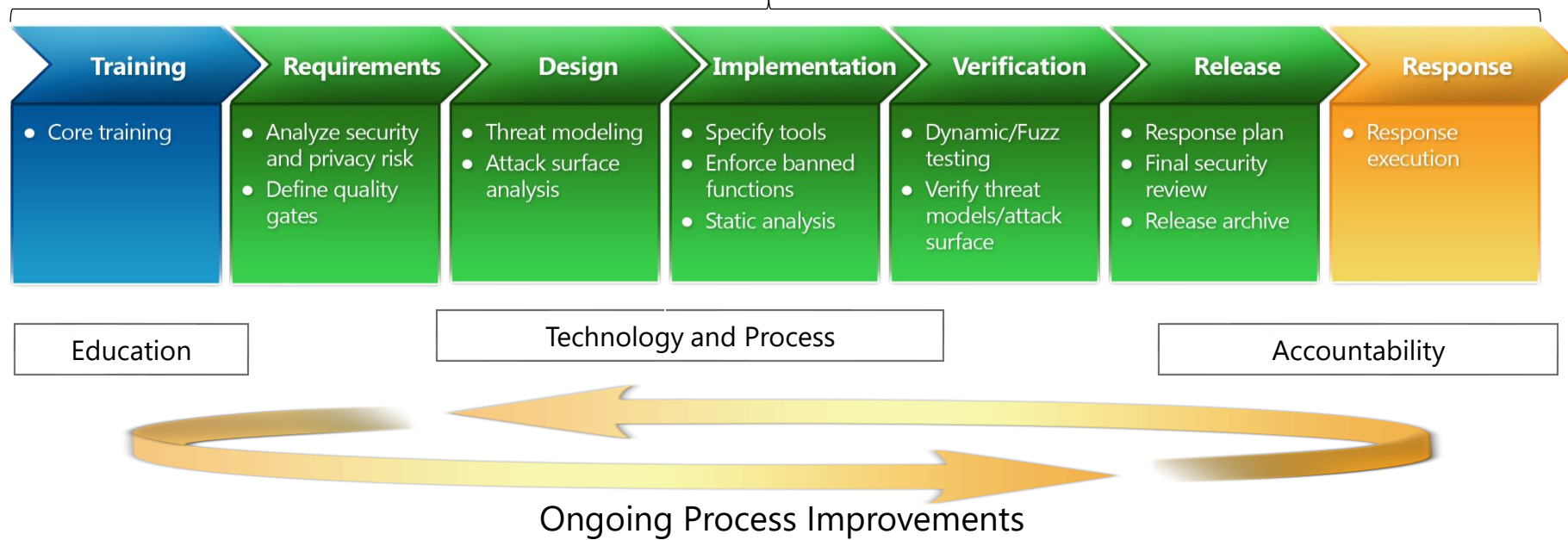
- This briefing is targeted at organizations that are getting started with SDL
 - New organizations with limited resources
 - Small organizations
 - Organizations that haven't paid attention to software security
- For bigger organizations that are big targets – perhaps a useful guide to getting started
 - But should expect to wind up doing everything

SDL History and Origins

- First there was Windows
- Then there was the Security Response Center
- Then there was Code Red, Nimda, UPnP
- Then there was Trustworthy Computing
- Then there was the Windows Server 2003 Security Push
 - Trained everybody
 - Deployed the tools we had
 - Told everybody to stop work, do security reviews, fix bugs
- Pre-ship “audit” confirmed effectiveness of what was done
- Effectiveness led to decision to create sustained process

- Initial SDL (v 2.0) derived from Windows Security Push
 - Train engineers
 - Identify things the engineers must do to create secure software (more about this later)
 - Distribute “must do” activities throughout development cycle
- Final Security Review was the successor to the “audit”
 - Did the engineering team do the SDL?
 - Any major problems left?
 - Not “test security in at the end”
- Updated SDL on a (more or less) regular cadence
 - New classes of vulnerabilities
 - New techniques for secure development

SDL Big Picture (or picture of a big SDL)



- Design
 - Secure Design Principles
 - Threat Modeling
 - Perform Architectural and Design Reviews
 - Develop an Encryption Strategy
 - Standardize Identity and Access Management
 - Establish Log Requirements and Audit Practices
- Secure Coding Practices
 - Establish Coding Standards and Conventions
 - Use Safe Functions Only
 - Use Current Compiler and Toolchain Versions and Secure Compiler Options
 - Use Code Analysis Tools To Find Security Issues Early
 - Handle Data Safely
 - Handle Errors
- Manage Security Risk Inherent in the Use of Third-party Components
- Manual Testing
 - Perform Manual Verification of Security Features/Mitigations
 - Perform Penetration Testing
- Automated Testing
 - Use Static Analysis Security Testing Tools
 - Perform Dynamic Analysis Security Testing
 - Fuzz Parsers
 - Network Vulnerability Scanning
 - Verify Secure Configurations and Use of Platform Mitigations
 - Perform Automated Functional Testing of Security Features/Mitigations
- Manage Security Findings
 - Define Severity
 - Risk Acceptance Process
- Vulnerability Response and Disclosure
 - Define Internal and External Policies
 - Define Roles and Responsibilities
 - Ensure that Vulnerability Reporters Know Who to Contact
 - Manage Vulnerability Reporters
 - Monitor and Manage Third-party Component Vulnerabilities
 - Fix the Vulnerability
 - Identify Mitigating Factors or Workarounds
 - Vulnerability Disclosure
 - Secure Development Lifecycle Feedback

SDL for Smaller/Newer Organizations

- First SDL processes were created at big companies
 - I've talked about Microsoft
 - Other SAFECODE member companies
 - Other large companies
- For a big company
 - You write a lot of code
 - Products are under heavy scrutiny by vulnerability researchers – and attackers
 - Security problems create intense customer pressure
 - Policy may be “do everything”
- For a small company
 - “Do everything” is probably unaffordable

Smaller Organizations Advantages and Disadvantages



- + Maybe less scrutiny
- + Maybe less critical to customers (or maybe not!)
- + Probably smaller, simpler product family
- May depend more on code other organizations write
- Probably smaller security team
- Probably less budget for security tools and training
- Maybe less management commitment to product security

- The developers are responsible for creating secure software
- The security team supports the developers
 - Training
 - Tools
 - Consultation
 - Validation (but not “penetration testing security in”)
- Rely on root cause analysis and continuous improvement
 - Know why vulnerabilities occur
 - Figure out how to prevent them
 - Update your process, tools, training

So What To Do?

- Following slides suggest measures to incorporate in a “starter” SDL
- These are in (my estimated, rough) priority order and show
 - What
 - Why
 - How (and resources)
- I’m assuming an established organization with shipping products that wants or needs to create an SDL
- I’m identifying discrete activities that go in your dev process “where they fit” – integrate as appropriate whether Agile, DevOps, Spiral...
- Your mileage will vary

- Be prepared to accept vulnerability reports
 - Have an email alias and a website, answer reports
 - Fix reported ***and related*** vulnerabilities timely and correctly
 - Do root cause analysis
 - Why did we have this vulnerability?
 - ***How can we adapt our development process to prevent prevalent vulnerabilities?***
- Response process is a key input to the definition of your SDL!
 - It's a good source of input on the state of your software security
 - It also helps build customer confidence even if you don't have an SDL
- ISO 29147 and ISO 30111 are valuable guides for response

- Have a way to know what you've done
 - Fixing individual code bugs
 - Executing parts of your SDL process
- If you don't know what you've done, you're flying blind
 - When things go wrong, you don't know why
 - You don't have a good idea what parts of your process are working – you may not even have a process
- The more integrated SDL tracking is with your development bug tracking, the better
 - Add the bug types and effects
 - There are some commercial tools for common tracking systems

Have a Rating System and Bug Bar



- Think about the severity of different kinds of vulnerabilities and know which are “must fix”
 - Would you delay shipping to eliminate this problem?
 - Security bugs get rated by severity in the tracking system
- Not all vulnerabilities are created equal
 - Impact and exploitability matter
- Impact will depend on the product and how it’s used
 - Simpler severity rating system is probably better
 - You will probably underestimate ease of discovery and exploitation
 - CVSS v3 and Microsoft bug bar are useful references

- Select secure third party code and be ready to deal with vulnerabilities in components
 - “Secure” based on CVE history, reputation, developer having an SDL process
 - Cases range from embedding a product to copying a snippet of source code
 - Have an inventory
 - Have a plan to evaluate and respond
- Nobody builds everything themselves any more
 - *This is potentially more important than dealing with vulnerabilities in code you write*
- See SAFECODE guide on third-party components
- Some commercial tools to help with component management

Do The Easy Stuff

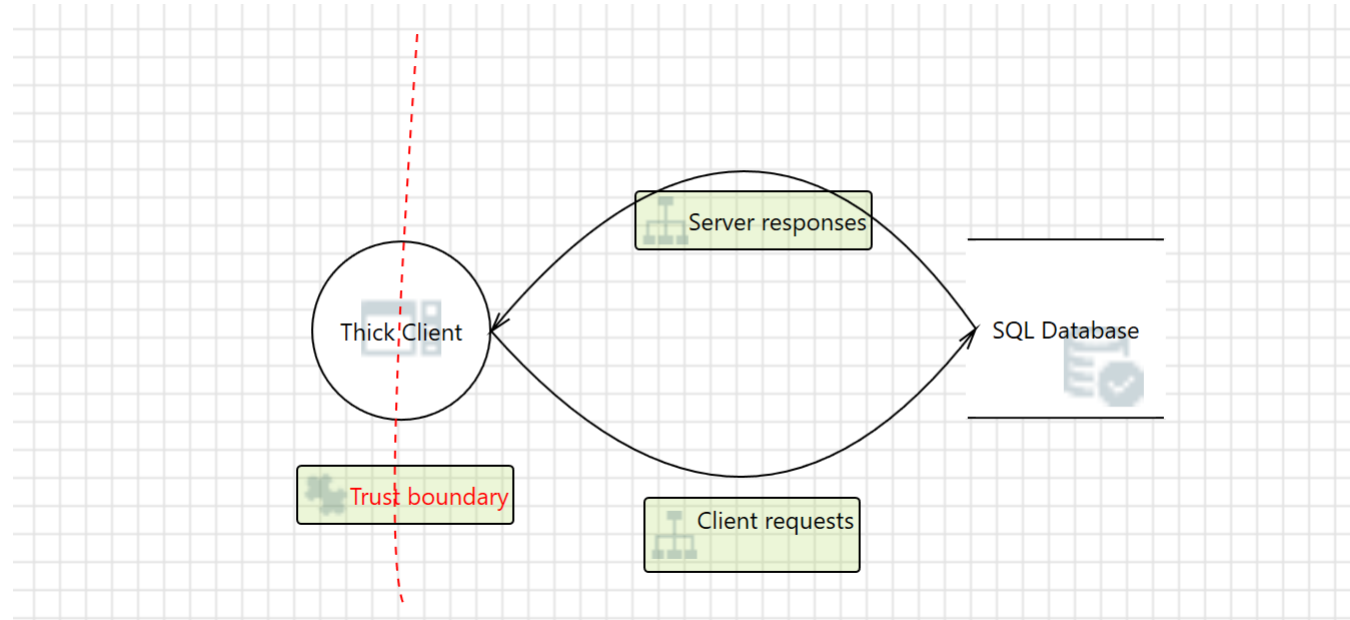
- Adopt secure development options that are “free”
 - Built into your tools
 - Little or no impact on developers, performance
 - These are no-brainers
- The secure development options are there because they address common vulnerabilities
 - Maybe you won’t experience them, but they’re *common*
- Examples
 - Building with compile and build time mitigations such as ASLR and DEP
 - Banning unsafe C/C++ APIs

- Not just the security team – get executive buy-in
- Get your engineering staff signed up to shipping secure code
 - It's 2018 – this shouldn't be necessary but...
 - “Why would anyone do that?”
 - “Tell them not to do that.”
- Developers must understand that product security is
 - Important
 - Their job
- Message has to come from someone they'll believe, and the message has to be serious

Train the Developers

- Once the developers are motivated, tell them what they need to do
 - Their college CS or SWE program didn't tell them ☹️
 - Training needs to be targeted to your developers, technology, products
- You're counting on the developers to produce secure software – not the security team
- Free and commercial resources are available (although targeted to your technology and product is better)
 - OWASP
 - SAFECODE

- Design or architect the product to protect itself and the users' data
- Designed-in bad security is expensive to fix, and probably impossible to fix quickly



- The best design analysis technique is threat modeling (see below) but copying a secure architecture may also be a practical way to start

- Don't roll your own security features if security isn't your business
- Many products require security features
 - Encryption
 - Identity and Access Management
 - Logging and Auditing
 - All are likely to be provided by platform or framework
- Use the platform features in a secure way
 - This is cheaper than rolling your own, and safer!

- “Attack surface” is jargon for exposed interfaces
 - Open ports
 - Unprotected files
 - Unauthenticated services
- If you close attack surface, you have to worry less about vulnerabilities in a component
 - Users still have to use your product – but there’s no reason to expose something that’s not used
- You can reduce attack surface with settings or code
- See the SDL book (free download) for more on attack surface
- Scanners can tell you what attack surface you’ve exposed

- Vulnerabilities from code-level errors are major source of problems
- Lots of languages and frameworks, lots of tools and techniques
- Some techniques (organizations with mature SDL do all)
 - Train your developers to avoid dangerous coding constructs
 - Use safe libraries (string handling, input and output sanitization, XSS safety)
 - Run code analysis tools
 - Run testing tools
 - Code-level penetration testing

- Don't get overwhelmed
 - Your technology narrows the problem (no buffer overruns in managed code)
- Response incidents and root cause analysis focus your effort
 - Train on the problems you (*or other companies like you*) have
 - All buffer overruns? You're already using safe libraries – probably time for static analysis and maybe fuzzing
 - All XSS? Maybe time for new libraries and testing
- Trying to do everything is great, but starting out, you need to prioritize

- Integrate security into development
 - Ideally, use desktop tools rather than run tools at the end
 - Integrate security tests into your pre-deployment tests
 - If you wait for the security team to run the tools and tests, the developers won't have time to fix the bugs – “we have to ship!” – this is critical for Agile and DevOps
- Select tools and enable tests cautiously
 - Your developers will hate false positives (Google Tricorder aims for 90% true positive)
 - Many tools take a lot of “training” for your code base and application
- Lots of tool options
 - Free/OSS (OWASP, Linux Foundation CII)
 - Free tool scans if you're building OSS code
 - Commercial products
 - Services (vendor runs the tool and gives you results)

- Threat modeling addresses design errors
 - Describe your design, then think about what can go wrong in a structured way
 - STRIDE vulnerabilities
 - Evaluate potential vulnerabilities and fix the important ones (see “Bug Bar”)
- Design level vulnerabilities are less common than code level, but they still occur
 - Can be very serious when they happen (see “Secure Design”)
 - Do a high level threat model first, then come back for details
- Lots of books, guides (SAFECode guide to tactical threat modeling), free and commercial tools, consultants

Some Final Basics

- Understand security before you ship or deploy
 - Whether you're releasing an OS every three years or an update to a website this afternoon
- The bug tracking system should have a bug for every requirement (and for investigating/fixing every tool find)
 - If the bugs are fixed, you've done the process
 - If not, it's time for a discussion about risk
- Penetration testing consultants?
 - Maybe – but know why you're using them
- I've listed verification late – that's where it sits in the schedule – but even if you only adopt a minimal SDL, you should verify that you've done it

- A bug bounty is a great complement to (or component of) a response process and SDL
- You pay external researchers for finding and reporting vulnerabilities
 - Great input to your root cause analysis and SDL updating
- It's not a substitute for having an SDL process
 - A good way to have the same experience over and over...
 - So if you're not doing at least some of the design and code parts of the SDL process, don't expect a bug bounty to add much to your product security

- SDL is an effective way to integrate security from attack into your software
- Full SDL process has a lot of parts, but adoption can be staged or tailored for affordability
- Principles are important
 - The developers build code that's secure
 - Root cause analysis drives your priorities
- A few basics to get started
 - Response process and root cause analysis
 - Management of third party component security

- BSIMM - <https://www.bsimm.com/>
- Howard and Lipner SDL book - https://blogs.msdn.microsoft.com/microsoft_press/2016/04/19/free-ebook-the-security-development-lifecycle/ (free)
- ISO 29147 - http://standards.iso.org/ittf/PubliclyAvailableStandards/c045170_ISO_IEC_29147_2014.zip (free)
- ISO 30111 - <https://www.iso.org/standard/53231.html> (paywall)
- Linux Foundation Core Infrastructure Initiative - <https://www.coreinfrastructure.org/programs/tooling/>
- Microsoft SDL v5.2 - <https://msdn.microsoft.com/en-us/library/windows/desktop/cc307748.aspx>
- Microsoft SDL Bug Bar - <https://msdn.microsoft.com/en-us/library/windows/desktop/cc307404.aspx>
- OWASP resources (documents, tools, training) - https://www.owasp.org/index.php/Main_Page
- SAFECODE documents - <https://safecode.org/publications/>
- SAFECODE training - <https://safecode.org/training/>