PA193 - Secure coding principles and practices

Security Code Review

Łukasz Chmielewski chmiel@fi.muni.cz

CROCS
Centre for Research on Cryptography and Security

https://drive.google.com/file/d/1KSlG9uI-BiavBrvODeiaXUge_ZeIrp2W/view?usp=sharing

PROBLEM

Example problem – Debian RNG flaw

- Linus's law
  - "Given enough eyeballs, all bugs are shallow"
  - https://en.wikipedia.org/wiki/Linus%27_Law
- Flaw in Debian's random number generator (2008)
  - CVE-2008-0166
  - http://www.debian.org/security/2008/dsa-1571
  - lead to predictable random numbers
  - improper change to OpenSSL random generator
  - persisted for almost two years!
  - lead to only 262148 possible openSSH keys
- Change made based on static and dynamic analysis tools recommendation!

Debian RNG flaw

- Valgrind and IBM's Purify reports problems
  - usage of uninitialized variable
  - OpenSSL crypto/rand/md_rand.c

```
MD_Update(&m,buf,j);
MD_Update(&m,buf,j); /* purify complains */
```

- Discussion of maintainers (before and after change)
  - http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516

Fatal mistake

Diff of /openssl/trunk/rand/md_rand.c

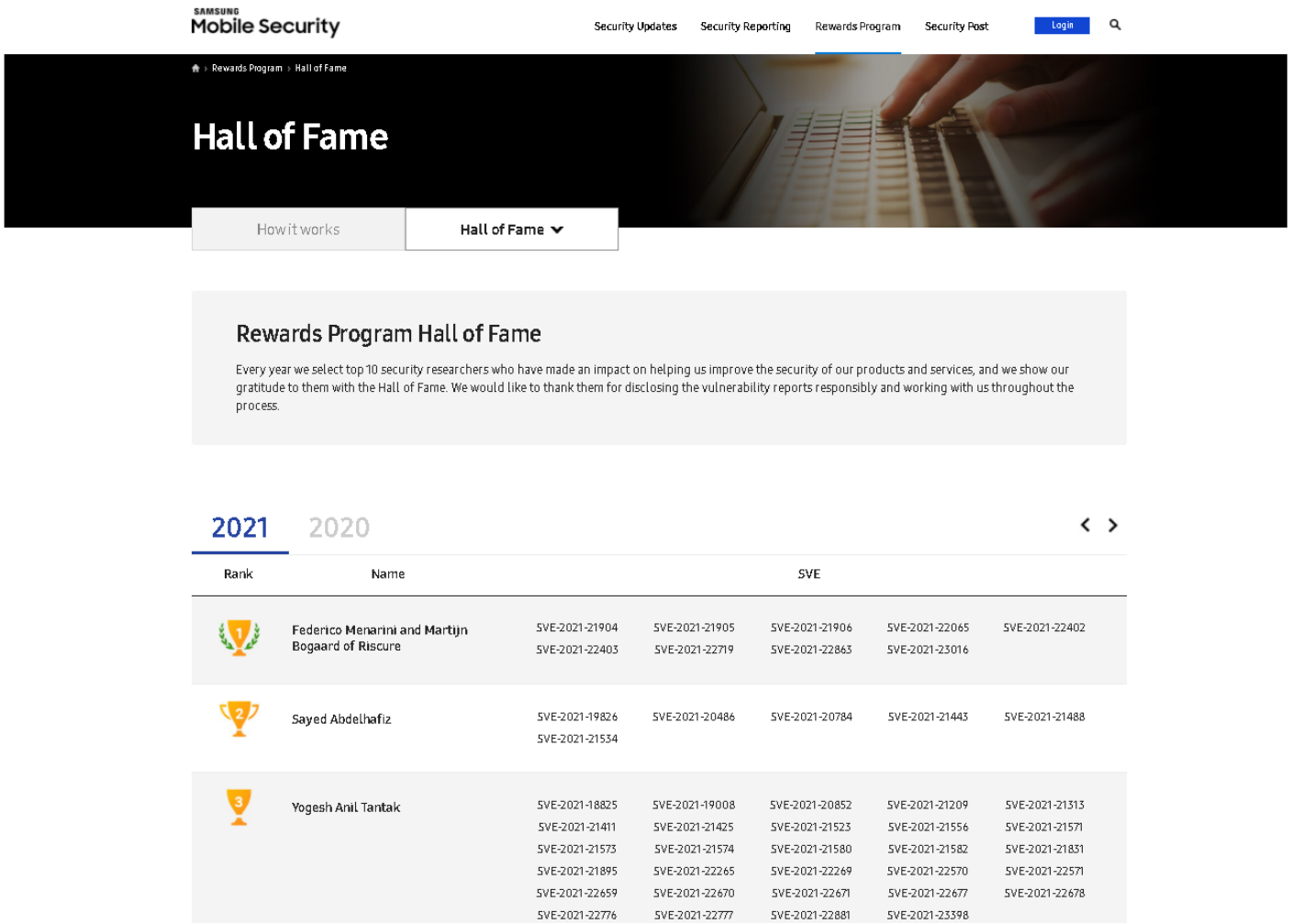
Parent Directory | Revision Log | Patch

#	revision 140 by *kroeckx*, Tue May 2 16:25:19 2006 UTC	revision 141 by *kroeckx*, Tue May 2 16:34:53 2006 UTC
#	**Line 271** static void ssleay_rand_add(const void *	**Line 271** static void ssleay_rand_add(const void *
271	else	else
272	MD_Update(&m,&(state[st_idx]),j);	MD_Update(&m,&(state[st_idx]),j);
273		
274		/*
275		* Don't add uninitialised data.
276	MD_Update(&m,buf,j);	MD_Update(&m,buf,j);
277		*/
278	MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));	MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
279	MD_Final(&m,local_md);	MD_Final(&m,local_md);
280	md_c[1]++;	md_c[1]++;
#	**Line 465** static int ssleay_rand_bytes(unsigned ch	**Line 468** static int ssleay_rand_bytes(unsigned ch
468	MD_Update(&m,local_md,MD_DIGEST_LENGTH);	MD_Update(&m,local_md,MD_DIGEST_LENGTH);
469	MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));	MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
470	#ifndef PURIFY	#ifndef PURIFY
471		/*
472		* Don't add uninitialised data.
473	MD_Update(&m,buf,j); /* purify complains */	MD_Update(&m,buf,j); /* purify complains */
474		*/
475	#endif	#endif
476	k=(st_idx+MD_DIGEST_LENGTH/2)-st_num;	k=(st_idx+MD_DIGEST_LENGTH/2)-st_num;
477	if (k > 0)	if (k > 0)

Morale

- Access to source code doesn’t guarantee bug-free code
- Usage of automated tools can provide great advantage, but deep understanding of code before change must remain
- Manual code review eventually spotted the problem

Why to perform source code review?



SAMSUNG Mobile Security

Security Updates | Security Reporting | Rewards Program | Security Post | Login

Rewards Program › Hall of Fame

Hall of Fame

How it works | Hall of Fame

Rewards Program Hall of Fame

Every year we select top 10 security researchers who have made an impact on helping us improve the security of our products and services, and we show our gratitude to them with the Hall of Fame. We would like to thank them for disclosing the vulnerability reports responsibly and working with us throughout the process.

2021 2020

Rank	Name	SVE				
1	Federico Menarini and Martijn Bogaard of Riscure	SVE-2021-21904	SVE-2021-21905	SVE-2021-21906	SVE-2021-22065	SVE-2021-22402
		SVE-2021-22403	SVE-2021-22719	SVE-2021-22863	SVE-2021-23016	
2	Sayed Abdelhafiz	SVE-2021-19826	SVE-2021-20486	SVE-2021-20784	SVE-2021-21443	SVE-2021-21488
		SVE-2021-21534				
3	Yogesh Anil Tantak	SVE-2021-18825	SVE-2021-19008	SVE-2021-20852	SVE-2021-21209	SVE-2021-21313
		SVE-2021-21411	SVE-2021-21425	SVE-2021-21523	SVE-2021-21556	SVE-2021-21571
		SVE-2021-21573	SVE-2021-21574	SVE-2021-21580	SVE-2021-21582	SVE-2021-21831
		SVE-2021-21895	SVE-2021-22265	SVE-2021-22269	SVE-2021-22570	SVE-2021-22571
		SVE-2021-22659	SVE-2021-22670	SVE-2021-22671	SVE-2021-22677	SVE-2021-22678
		SVE-2021-22776	SVE-2021-22777	SVE-2021-22881	SVE-2021-23398	

SECURITY CODE REVIEW

Resources

- Review process and techniques are extensively based on the excellent book "*The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*" by Mark Dowd, John McDonald, Justin Schuh
- Book is available in faculty library

Security code review

- Architecture overview
  - Design choices and possible design flaws
- Code review
  - How well is architecture actually implemented
- Whitebox, greybox & blackbox testing
  - different level of access to code and documentation
- Available tools
  - mainly for code review
- Certifications
  - Independent labs

Application review phases

1. Pre-assessment
2. Application review
3. Documentation and analysis
4. Remediation support

Timeline

- Good reviewer ranges between 100 to 1,000 lines of code an hour
  - highly dependent also on code complexity
  - flexibility must be allowed
- Keep track of your previous progress
  - and get feeling for your speed
  - helping you making better future estimations

Information Collection

- Developer interviews
- Developer documentation
- Standards documentation
- Source profiling
- System profiling

- Reports from previous reviews
- History of changes
  - For example, git log

Common problems

- Design documentation not available at all
- Design documentation is outdated
- Third party components without documentation
- Developers not available or not cooperating
- Limited time for everything

Iterative process

1. Plan your next work
2. Perform auditing strategy you selected
   – and make extensive notes
3. Reflect on time spend
   – what you have learned
4. Repeat from step 1.

Top-down approach

- Top-down approach
  - water-fall like approach
  - start from design specification
  - establish threat model
  - find design vulnerabilities first
  - find logical implementation vulnerabilities second
  - find low-level implementation bugs third
- Good results if design documentation is accurate
  - but that is usually not the case
  - something is missing or implemented differently

Bottom-up approach

- Bottom-up approach
  - starts with implementation
  - targets low-level implementation vulnerabilities first
    - e.g., by automated tools
  - higher-level threat and design documentation later
    - when understanding of application is much better
- Works well even if design documentation is not accurate
  - but is slow as you need to read a lot of code that is NOT security relevant
- Necessity for maintaining design model continuously
  - e.g., DFD sketches and class diagrams

Hybrid approach

- Combination of top-down and bottom-up approaches
- Focus on high-level characteristics
  - General application purpose
  - Assets and entry points
  - Components and modules
  - Inter-module relations
  - Fundamental security
  - Major trust boundaries

Verify the progress you are making

Ask often following questions:

- *What have you learned about the application?*
- *Are you focusing on the most security-relevant components?*
- *Have you gotten stuck on real problem or gone down some rabbit hole?*

Verify the progress you are making (cont.)

Ask often following questions:

- *Does your master ideas list have many plausible entries?*
- *Have you been taking adequate notes and recorded enough detail for review purposes?*
- *If you're working from application models and documentation, do these models reflect the implementation accurately?*

Security code review - hints

- You will always have a limited time
  - try to rapidly build overall picture
  - use tools to find low hanging fruit
- Focus on most sensitive and problematic areas
  - use tools to focus your analysis scope
- More eyes can spot more problems
  - experts on different areas
- It’s creative process
  - be pragmatic, flexible, and results driven
- Have the rights skills
  - you should know programming as well as have security mindset

Present results (Finding summary)

- Location of the vulnerability
- Vulnerability class
- Vulnerability description
- Prerequisites (for exploiting vulnerability)
- Business impact (on assets)
- Remediation (how to fix)
- Risk
- Severity
- Probability

Finding summary - example

Problem identification: DSA-1571-1 openssl
Severity: critical
Risk: high - directly exploitable by external attacker
Problem description: crypto/rand/md_rand.c:276 & 473 – The random number generator in Debian's openssl package is predictable. This is caused by an incorrect Debian-specific change to the openssl package. One of the sources of a randomness based on usage of uninitialized buffer *buff* is removed.
Remediation: revert back to usage of uninitialized buffer *buff*

Architecture review

Architecture overview

- Get all information you can quickly
- Assets
  - What has the value in the system?
  - What damage is caused when successfully attacked?
  - What mechanisms are used to protect assets?
- Roles
  - Who has access to what?
  - What credentials needs to be presented?
- Thread model
  - What is expected to do harm?
  - What are you defending against?

Architecture review (2)

- Usage of well established techniques and standards
- Comparison with existing schemes
  - What is the advantage of new scheme?
  - Why changes were made?
- Security tradeoffs documented
  - Possible threat, but unmitigated?
  - Is documented or overlooked?

Sensitive data flow mapping

- Identify sensitive data
  - password, key, protected data...
- Find all processing functions
  - and focus on them
- Create data flow between functions
  - e.g. Doxygen call graph
- Inspect when functions can be called
  - Is key initialized?
  - Can be function called without previous function calls?
- Where are sensitive data stored between calls?

Protocol design (and implementation)

- Packet confidentiality, integrity and authenticity
- Packet removal/insertion detection
- Replay attack
- Reflection attack
- Man in the middle

Cryptography usage

- CIA (Confidentiality, Integrity, Availability)
  - Plaintext data over insecure channel? Encrypted only?
  - Can be packet send twice (replay)?
  - What is the application response on data modification?
- What algorithms are used
  - Broken/insecure algorithms? MD5? simple DES?
- What key lengths are used?
  - < 90 bits symmetric crypto?
  - < 1024 bits asymmetric crypto?
- Random number generation
  - Where the key comes from?
  - Is source entropic enough?
  - srand() & rand()?

Cryptography usage (2)

- Key creation
  - Where the keys originate? Enough entropy?
  - Who has access?
- Key storage
  - Hard-coded keys
  - Keys in files in plaintext
  - Keys over insecure channels
  - Keys protected by less secure keys
  - Key in Trusted Elements / Hardware
- Key destruction
  - How are keys erased from memory?
  - Can exception prevent key erase?

Cryptography implementation

- Implementation from well known libraries?
- Own algorithms?
  - security by obscurity?
  - usually not secure enough
- Own modifications?
  - Why?
  - sometimes used to prevent compatible programs
  - decreased number of rounds?
  - Performance optimization with security impact?

CODE INSPECTION

Example process

1. Start review by suite of static analysis tools
   - approximately up to 40-50% of software bugs can be found
   - but incapable of finding application flaws and business logic vulns.
2. Results used to create prioritized list for human review
   - security mechanisms to review
   - potential security vulnerabilities to investigate
3. Manual inspection of issues in prioritized list
   - use and abuse cases
   - various code inspection strategies
4. Threat modeling used for large codebases (>100k loc)
   - inspect impact of generally high-risk threat on application

- https://web.archive.org/web/20150329111329/https://www.praetorian.com/campaign/software-security/security-code-review.html

Code navigation

- Control-flow sensitive navigation
  - follow function calls
  - e.g., what parts of program are reachable from set of functions callable without previous authentication?
- Data-flow sensitive navigation
  - follow flows of interesting data
  - e.g., password from input to verification and storage
- Code navigation tools provide great help
  - call graphs (Doxygen, Performance profilers)
  - tainted values (e.g., taintgrind)
  - ...

Code auditing strategies

- Code comprehension (CC) strategies
  - analysing the source code directly to discover vulnerabilities
- Candidate point (CP) strategies
  - create a list of potential issues (via some mechanism)
  - examine the source code for relevance of these issues
- Design generalization (DG) strategies
  - reviewing the implementation and inferring higher-level design abstractions
  - medium- to high-level logic and design flaws

Code comprehension (CC) strategies

CC strategy - Trace Malicious Input

- Start at entry point to the system
  - e.g., user input
- Trace flow of code forward with data flow analysis
  - functions processing user input
- Set of possible “bad” inputs is created
  - e.g., escaped shell command
- Code is examined for potential security issue
  - where is user input “executed”?

Trace Malicious Input - characteristics

Start point	Data entry points
End point	Security vulnerabilities (open-ended)
Tracing method	Forward, control-flow sensitive, data-flow sensitive
Goal	Discover security problems that can be caused by malicious input. Use threat model and/or common vulnerability classes to help guide analysis.
Difficulty	Hard
Speed	Very slow
Comprehension impact	High
Abstraction	Basic implementation through implementation logic
Strengths	Inherent focus on security-relevant code Can sometimes identify subtle or abstract flaws Difficult to go off track
Weaknesses	Code and data paths balloon up quickly, especially in object-oriented code Easy to overlook issues Requires focus and experience

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities

CC strategy - Analyse Module & Algorithm

- Reading the code line by line from the beginning
- Do not follow function calls
- Writing down potential issues spotted

- Algorithm analysis is similar to module analysis, but module implementation is usually longer
- Effective, if the code is not too long, but mentally exhausting
  - overlooked problems after some time, time-demanding

Analyse a Module - characteristics

Start point	Start of a source file
End point	End of a source file
Tracing method	Forward, not control-flow sensitive, not data-flow sensitive
Goal	Look at each function in a vacuum and document potential issues.
Difficulty	Very hard
Speed	Slow
Comprehension impact	Very high
Abstraction	Basic implementation through design
Strengths	You learn the language of the application
	Easier to analyze cohesive modules
	Can find subtle and abstract flaws
Weaknesses	Mentally taxing
	Constant documentation requires discipline
	Easy to mismanage time

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities

CC strategy – other useful strategies

- Analyse a Class or Object
  - implementation of small unit
- Trace Black Box Hits
  - focus on areas where fuzzers etc. found problems
  - e.g., by debugging with value used to crash application
- Automated Source Analysis Tool
  - used to generate candidate points

Candidate points (CP) strategies

Candidate points strategies

1. Use some tool or process for identifying candidate points
2. Deeper follow-up inspection by other (e.g., CC) strategy

- Simple Lexical Candidate Points
  - patterns of common vulnerabilities (full text search, grep-like tool)
  - deprecated functions (e.g., gets), strings like “key”, “password”...
  - static analysis tools, e.g., Cppcheck rules

Candidate points strategies

- Simple Binary Candidate Points
  - generate candidate points from binary only (unavailable source code)
  - list or search for specific strings in binary
  - search for interesting system calls
  - use disassembling, or binary debugging

- Application-Specific Candidate Points
  - patterns of mistakes for particular application
  - learned from previous code/binary analysis
  - e.g., new rule for Cppcheck

TOOLS

Handy tools

- Syntax highlighting, full text search
  - any reasonable editor
- Regular expression tools (grep)
  - allow for more complex searches
- Automatic generation of call graphs
  - Doxygen, Visual Studio and many other tools

Handy tools

- Static and dynamic analyzers
  - detect multiple issues
  - annotations (e.g., SAL) will help even further
- Fuzzing tools
  - behavior under stress, error messages...
- Mind-mapping software
  - build and do not forget information you got
- Tools to find different instances of existing issues
  - CodeQL
- Pen&Pencil
  - still of great help (flexible)

ANTI-PATTERNS

(Security) Antipatterns

- Common defective process and implementation within organization
- Opposite to design patterns
  - see http://sourcemaking.com/design_patterns
- Read http://sourcemaking.com/antipatterns
  - good description, examples and how to solve
  - not limited to object oriented programming!

Security anti-patterns

- Software development anti-patterns
  - http://sourcemaking.com/antipatterns/software-development-antipatterns
- Tesco password handling
  - http://www.troyhunt.com/2012/07/lessons-in-website-security-anti.html
- Critique of some usages of OAuth
  - http://adactio.com/journal/1357/

Recommended reading

- Process of security code review
  - https://ieeexplore.ieee.org/document/1668009
- Software Security Code Review
  - https://web.archive.org/web/20100208065233/http://www.softwaremag.com/l.cfm?doc=2005-07/2005-07code
- Performing security Review (Microsoft)
  - https://web.archive.org/web/20141023191707/http://silverstr.ufies.org:80/blog/msdn-webcast-code-review.pdf
- SDL security code review process (MS Security Push)
  - http://msdn.microsoft.com/en-us/library/cc307418.aspx
- OWASP security review
  - https://www.owasp.org/index.php/Security_Code_Review_in_the_SDLC
- On the effectiveness of code review
  - https://mfinifter.github.io/papers/coderev-essos13.pdf

Recommended reading

- Why cryptosystems fail, R. Anderson
  - http://www.cl.cam.ac.uk/~rja14/Papers/wcf.pdf
- Static code analysis tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
- Security in web applications (OWASP)
  - http://www.owasp.org/index.php/Code_Review_Introduction
- How to find 5 RCEs in Apache Struts with CodeQL
  - https://securitylab.github.com/research/apache-struts-CVE-2018-11776/

CONCLUSIONS

Conclusions

- Plan your work and time (work iteratively)
- Different reviews needs different techniques (be flexible)
- Code review is creative process (have fun)
- Tools can help you a lot (use them)
  - but main part of work is up to you
- Code review also contains human interaction (be polite)

Questions ?