Statically Scanning Java Code: Finding Security Vulnerabilities
http://ieeexplore.ieee.org/abstract/document/6156713/

- Creates a tool called Jslint that statically scans Java source code for "potentially" insecure coding practices
- Goal is to identify and correct potential security problems during software development process (instead of in the field patching)
- The proposed flow of software design in the paper is: (seems to follow the waterfall model…)
    - Design a system with security in mind
    - Analyze the system, knowing the risks and anticipated risks
    - Rank the risks in order of severity (e.g. password leak more severe than intruder)
    - Test the risks
    - Cycle the broken system back through the design process
- The want is to minimize the "penetrate and patch" approach
- "Finding and removing bugs in a software system before its release is orders of magnitude cheaper and more effective…"
- Problem with the approach is that most software developers are unfamiliar with software security techniques and attacks
- Paper developed 12 rules for more secure Java code and Java itself is already outfitted for security.
- Java currently exposes you to 3 risks (paper found this from the 12 rules it compiled):
    - A JVM flaw may allow a malicious java program to breach the JVM security
        - Solved by securing the Java platform
    - User may mistakenly grant powerful privileges to malicious code
        - Only user good judgment can prevent this
    - User might grant privileges to bugged code that then malicious code can exploit
        - What Jslint wants to solve
- Jslint does not want to guarantee security but eliminate certain kinds of security attacks (make impossible for those attacks but other kinds of attacks are possible)
- The 12 rules:
    - Don't depend on initialization
        - Paper states to not use a constructor to initialize but isn't that a bad practice if you don't do that??? (especially when you're making multiple instances of an object)
    - Limit access to your classes, methods, and variables
        - Document reasons for why a method is public because it should only be public for a good reason
    - Make everything final by default, unless there is a good reason not to do so
        - Don't allow overriding of methods because an attacker can extend the method in a malicious way
    - Don't depend on package scope
        - Similar to #2 but mainly because packages can be opened up and a new class can get added. You can try closing the package but better off assuming not possible to close
    - Do not use inner classes (nested classes)

        - Inner classes can actually be accessed by classes outside of the wrapping class by manipulating the byte code.
        - The inner class can also cause more damage since it can access the fields of the wrapping class regardless of private
    - Avoid signing your code
        - Signed code gives your code privileges than normal but the problem is your signature can get stolen and used by malicious code to get access to very dangerous privileges! Better if your code is treated as an outsider to the user computer than a close friend
    - If you must sign your code, put it all in one archive file
        - By sealing up your code by signing the multiple classes with a single signature as if it were a single class, attackers cannot mix and match your code with their malicious code
    - Make your classes uncloneable
        - Clone is when you allow instance creation/duplication without calling the constructor. You want to prevent this because if the object is a very "dangerous" object => attacker can use the clone to do his bidding instead. If you need cloning ability, define a clone method and make it final or override parent's clone method and make it final. Otherwise, leave it out completely.
    - Make your Classes Unserializable
        - Don't save the state of the object into disk etc. for later reference. (Hackers can then manipulate your memory to change the object)
    - Make your classes Undeserializable
        - Even though you make your class unserializable, it might still deserializable (readable from memory).
    - Don't compare classes by name
        - Use equality of super etc. instead (or methods that grab the class and can compare the classes directly)
    - Secrets stored in your code won't protect you
        - Don't store keys in your code (don't hardcode the keys basically)
        - Code obfuscation is a way to store a secret in your code (though I disagree that it's not possible to read… there's lots of code deobfuscation tools out on GitHub these days which are needed to reverse engineer apps/inner workings)
- JSlint only covered 8 of the rules for auto correction (1-4 and 8-11)
- Uses static scanning instead of dynamic scanning because it aims to find the hostile code before running
- Attackers can easily get around the static scanning by knowing what the scanner will detect
- Use the tool during code development instead of when released => no point of attacker coming in yet
- JSlint first parses the source file into a syntax tree and uses the Visitor design pattern.
    - Very similar to JDiff but instead of finding differences in the code, parses each Visitor (the similar equivalent of a hammock) to check for security violations

    - Also allows users to define new scanning criterion by defining a new class (extension of the original scanning criterion)

Security band-aids: more cost-effective than "secure" coding
http://ieeexplore.ieee.org/abstract/document/1049389/

- Companies often inherit applications that have their own unique problems (including security) so there's no point in hunting down engineers to pin the blame of unsecure code
- "Although the right place to solve software problems is in development, most companies do not have the luxury of time or money to rebuild old code"
- Leads to patches which are the cheapest and easiest way possible (not best)
- Timesaving point solutions like firewalls have an instant return but are not good in the long run since they don't prevent the problem from coming in another way!
- There's also a human element where developers are forced to deliver new projects on time with a short deadline thus leading to them throwing their code to the security team which analyzes the code after the fact
    - Lots of stuff are slid through
- The developer world is separate from the true universe of code that meets production => lots of threats and unknowns.
    - Engineers are more like construction workers where they build it and get more money by getting certifications.
    - Certifications do not mean that the developer won't write bad code!
- Quality assurance testing doesn't catch all of the security holes as well. (there's only so much a QA engineer can do when faced with so many bugs and tooling issues)
- Band-aid security does not fix the disease but the protect the wound
    - Detect the bad but do nothing to stop the threat of an unknown attack
    - Basically, leave the possibility of getting infected open but once infected don't let anyone hurt on the same spot.
    - Best because there's always new threats and we can't anticipate those threats.
- Band-aid security involves using shunts and limiters on data input
- Example viruses where band-aid security helps against them: Nimda and Code Red
- Band-aids do not prevent a single attacker but are effective against mass attackers and automated attacks
- "Building more secure software is a goal, but it won't stop the virus that gets released tomorrow"

Secure Coding: Building Security into the Software Development Life Cycle
http://dx.doi.org/10.1201/1086/44797.13.5.20041101/84907.5

- Believes security has to be baked in to the overall systems development life-cycle
    - Unsophisticated software development techniques, a lack of security focused QA, and scarce security training for software developers are the culprits of side stepped security practices
- Lots of government laws & councils have been created to try and pinpoint security practices that should be incorporated into the development cycle
    - The results are more like recommendations => they do not directly address the root of the breaches: the failure of developers to take a security view of the product from inception, deployment, and beyond.

- To implement the security controls has become more complex and challenging in the past decade.
    - Traditional boundaries around organizations are blurring because of the limitless possibilities of commerce via web-based supply chains => firewall systems have become ineffective
- Security properties that are repeatedly outlined in the regulations include: accountability, unique user accounts, and confidentiality
    - Each property can be easily circumvented by software developers when not paying attention
- Information security is an afterthought for many organizations => not woven into IT projects => only after the system has been designed, tested, and ready for deployment do they test security
    - Mainly because:
        - Security is not considered a business enabler (or revenue generator)
        - Without upper management support, security dies ☹
        - Developers are hired for their coding expertise and not for their security knowledge. Nor are they given the training, tools, time, etc. to build secure systems
        - Software project deadlines end up causing primary effort to be on the features rather than the security
        - Security is often perceived as a barrier to functionality
- Paper states that security should exist at every portion of the software development section:
    - At design time, a security lead must be present to identify the possible security issues. The security lead makes it aware to the team that the ultimate goal of the development may expose certain security holes.
    - As the designing goes on, the security lead must also do threat modeling. Threat modeling involves the creation of the application model and assessing the landscape of the application to clearly identify the risks
    - At the coding stage (development stage), the developer must have the following practices:
        - Validate user input
        - Always assume a hostile environment
        - Use Open Standards
        - Minimize and Protect the trusted components in the system
        - Always authenticate the user (never trust that the user is pre-authenticated)
        - And more (very long list in the paper)
    - Testing stage requires unit tests that check for the security as well!
- Overall summary of the paper: Security practices need to be in place for the software development life but they use a person to pinpoint these issues (along with quality assurance tools) => utilize management and training to prevent security issues.

DLint: dynamically checking bad coding practices in JavaScript
http://dl.acm.org/citation.cfm?id=2771809

- DLint = dynamic checker

- Presents 28 different checkers that DLint will run (tries to state a dynamic checker is better than the static checker for JavaScript & possibly in general?)
- Compares against JSHint (the existing Static JavaScript checker)
- Other than security checking, also does bug checking (code quality checks)
- Final conclusion was that you need a variety of both static and dynamic
- The paper defines what it considers as a code quality rule, runtime event predicate, and runtime patterns
- Problems it notes that could contribute to poor code quality:
    - Inheritance: Inconsistent constructor and shadowing prototype properties (the latter being unique to JavaScript though)
    - Types: “undefined” type and concatenating an “undefined” with a String
    - Language Misuse: (mainly items available in javascript)
    - API misuse: (also specific to JavaScript ish…)
    - Uncommon Values: Not A Number value (NaN) should not be dependent on => allows a number which may become a problem
- Paper apparently found a correlation that popular websites have fewer violations of code quality rules (note that this is not causation!)