Notes

# Exam

* Explain how simple buffer overflows work & what root causes are
* Spot a simple buffer overflow, memory-allocation problem, format string attack, or integer overflow in some C code
* Explain how countermeasures - such as stack canaries, non-executable memory, ASLR, CFI, bounds checkers, pointer encryption, guards pages, etc ... - work
* Explain why they might not always work
* Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* Graphical user interface, table

  Description automatically generated with medium confidence
* Table

  Description automatically generated with medium confidence
* Table

  Description automatically generated
* Graphical user interface, text, application, email

  Description automatically generated
* Graphical user interface, text, application, email

  Description automatically generated
* Graphical user interface, text

  Description automatically generated

# Strings Considered Harmful

<https://www.usenix.org/system/files/login/articles/login_winter18_05_poll.pdf>

## LangSec and Parsing Flaws

* Root Problems
  + the sheer number of input languages that a typical application handles;
  + their complexity;
  + their expressivity;
  + the lack of clear, unambiguous specifications of these languages;
  + and the handwritten parser code
* Leads to bugs and buffer overflows in processing file formats such as Flash.

## Forwarding Flaws

* Large class of flaws involves the careless forwarding of malicious input from front-to-back-end where the input is then incorrectly verified.
* examples are:
  + format string attacks,
  + SQL injection,
  + command injection,
  + path traversal,
  + and XSS.
* Forwarding flaws are also called injection flaws, e.g., in the OWASP Top Ten, where they occupy the first spot. We prefer the term “forwarding flaws” because in some sense all input attacks are injection attacks; the forwarding aspect is what sets these input attacks apart from the others.

## Anti-Patterns

* Input Sanitization: Input sanitization is highly context sensitive, input may need different sanitization for HTML than it does for SQL, for example. This means that having a singular sanitization function is “suspect” and should never be done.
* String Concatenation: String concatenation is a form of “unparsing”.
* Strings: Heavy use of strings can cause trouble because it is a generic data type:
  + Strings can be used for a plethora of data, usernames, email, URL, etc.
  + By definition unparsed data.
  + Often brings unwanted expressivity which leads to unwanted functionality.

## Remedies

* Reducing Expression Power: For example, parameterized queries in SQL.
* Types to Distinguish Languages and Formats: Different types of URLs, usernames, etc. Data may need to be serialized/deserialized if it is sent to a different component.
* Types to Distinguish Trust Levels: Using different types for user submitted data and for hard-coded data, even though it is fundamentally the same type of data, a TrusedString can be made and a normal String can be made. Used by Google to combat XSS.

## Beyond Types: Programming Language Support

Wyvern, no clue what it does, hope it’s not on the exam.

# Log4Shell

The Log4Shell exploit affects an open-source Java tool called log4j which is widely used for application event logging;

Log4Shell is all the more serious for being relatively easy to exploit. The vulnerability works by tricking the application into interpreting a piece of text as a link to a remote resource, and trying to retrieve that resource instead of saving the text as it is written. All that’s necessary is for a vulnerable device to save the special string of characters in its application logs.

The log4j vulnerability was [first spotted in Minecraft](https://arstechnica.com/information-technology/2021/12/minecraft-and-other-apps-face-serious-threat-from-new-code-execution-bug/) servers, which attackers could compromise using chat messages; and systems that send and receive other message formats like SMS clearly are also susceptible.

# LangSec (Language-theoretic security)

<http://langsec.org/bof-handout.pdf>

## Mission Statement

* Design and Programming philosophy
* Formally correct and verifiable input handling
* Practical method of assurance
* Related to incorrect parsing and interpretation of messages between software components (packets, file formats, etc.)
* Recognizer rejects non-conforming inputs and transforms conforming inputs to structured data
* Aims:
  + Produce verifiable recognizers
  + Produce verifiable, composable implementations of distributed systems without exploits (Regarding messages)
  + mitigate the common risks of ungoverned development by explicitly exposing the processing dependencies on the parsed input
* correctness and computational equivalence of input processors

## Root Causes

* The existence of exploitable bugs is a consequence of software designs that make verification and comprehensive testing infeasible and undecidable in the formal sense.
* Exploits expose and use unintended states
* Exploit computation can be modeled as a computation occurring on an automaton whose states and transitions are supersets of the programmer’s intended model.
* Bug in input processing dominates other kinds of bugs.
* Input is often checked with recognizers that only check eg. Regex and not all other properties and the rest of the code assumes those were checked.
* Handling of inputs must be the first target of verification.

1. Ad-hoc notions of input validity
   1. Formal language-theoretic specification
   2. Should be optimized
2. Parser differentials: mutual misinterpretation between system components.
3. Mixing of input recognition and processing (a.k.a. “Shotgun parsers”)
4. Ungoverned development: Adding New Features / Language Specification Drift

# Secure Software Lifecycles

<https://www.cybok.org/media/downloads/Secure_Software_Lifecycle_v1.0.2.pdf>

Secure software lifecycle processes are proactive approaches to building security into a product.

## Microsoft Security Development Lifecycle (SDL)

1. Provide Training
   1. Provide CyberSecurity training to professional developers
2. Define Security Requirements
3. Define Metrics and compliance reporting
4. Perform threat modelling
   1. Spoofing: Posing as another user/valid server
   2. Tampering: Malicious modification of data
   3. Repudiation: Deny performing without the possibility of proof
   4. Information Disclosure: Exposure of information to individuals that should not see it
   5. DoS: Making system unavailable
   6. Elevation of privilege: Gain privilege and compromise the system
5. Establish design requirements
   1. Economy of mechanism: Keep it as small and simple as possible
   2. Fail-safe defaults: White-list instead of black-list
   3. Complete mediation: Every access to every object should have authenticate check
   4. Open design: Only key/password has to be secret
   5. Separation of privilege: A protection mechanism that requires two keys to unlock is more robust than one that requires a single key when two or more decisions must be made before access should be granted.
   6. Least privilege: Only use as much privilege as needed
   7. Least common mechanism: Minimize the amount of mechanisms common to more than one user and depended on by all users.
   8. Psychological acceptability: The human interface should be designed so that the user can easily and routinely apply the mechanisms correctly and securely.
   9. Defense in depth: provide multiple layers of security controls for redundancy
   10. Design for updating: The software security must be designed for change.
6. Define and use cryptography standards
7. Manage the security risk of using third-party components
8. Use approved tools
9. Perform static analysis security testing (SAST)
10. Perform dynamic analysis security testing (DAST)
11. Perform penetration testing
12. Establish a standard incident response process

## Touchpoints

McGraw uses the term touchpoint to refer to software security best practices which can be incorporated into a secure software lifecycle. This is in order of effectiveness.

1. Code Review (Tools)
   1. SAST and manual code review
2. Architectural Risk Analysis
   1. Attack resistance analysis: Checklist/systematic approach of considering each system component relative to known threats.
   2. Ambiguity Analysis: Capture creative activity required to discover new risks. Requires 2 or more experts.
   3. Weakness Analysis: Third-party components.
3. Penetration Testing
4. Risk-based Security Testing
5. Abuse Cases
6. Security Requirements
7. Security Operations

## SAFECode

The Software Assurance Forum for Excellence in Code (SAFECode) is a non-profit dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods.

1. Application Security Control Definition
2. Design
3. Secure Coding Practices
4. Manage Security Risk Inherent in the Use of Third-Party Component
5. Testing and Validation
6. Manage Security Findings
7. Vulnerability Response and Disclosure
8. Planning the Implementation and Deployment of Secure Development