CS 4337 — Programming Language Paradigms, Notes

https://iniyanjoseph.com | iniyanijoseph@gmail.com

Software security (engineering software in a way it continues correctly under malicious attacks) is NOT security software (firewalls, encryption, etc).

Software must deal with

1. Complexity, inadequacy, and change
2. Incorrect assumptions
3. Poor implementation
4. Inadequate knowledge of security
5. Unintended, unexpected interactions
6. Lack of consideration of security

Pointers: A pointer is a variable pointing to a location in memory (containing data). It has a special value, NULL (does not point to any data) Two operations: Assignment: `&x` gets the location in memory for a specific variable Dereference: `*x` follows the pointer to a location in memory (the value of the variable)

This means you can do `*(&x+1)` to get another location in memory.

`malloc(size);` provides a way to create new memory in the heap dynamically, and `free(pointer);` provides a way to remove used memory in the heap.

Two problems:

Dangling pointers point to a freed heap-dynamic variable. This is dangerous, since it points to something which should not exist. Lost heap-dynamic variable is when a pointer is not accessible to be freed (garbage/memory leak, and is inefficient in use of memory).

A consequence of pointer arithmetic is that you can access data outside of the bounds of an array `*(p+5)` is equivalent to `array[5]`, where `float* p=array;`

That means you can access `array[len]` and more!

This is called "stack smashing/buffer smashing", and is the starting point for hacking

—

Let's see how you can check for safety

Type checking is ensuring that all the operands of an operator/function are of compatible types. A compatible type is one which has a type which is either directly allowed to be used, or can be coerced (implicitly converted) to an allowed type.

A type error is the application of an operator to an operand of incompatible type.

Code -> {Static Type Checking, Dynamic Type Checking} Static Type Checking -> {Explicit declaration, Implicit Declaration}

Static = Checked at compile time

- Explicit = Types are explicitly stated in code (i.e. int x, float y)
- Implicit = Types are "figured out" by the compiler (i.e. var p)

Dynamic = Checked at runtime

—

Strongly typed if it enforces strict rules about how types are used. This is SEPERATE from type checking. This allows data to be used safely, but there is less flexibility with how data is used.

C and C++ are not strongly typed (any type can be used as another type, without coercion)

A programming language is type safe if the only operations allowed to be performed on data are those allowed by the data's type. This can be done at either compile time or run time.

Strong vs Weak safety Strong vs Weak refers to the amount of restriction on what operations allow types to mix. int*double? string+int? etc.

- It is weakly typed if it is both typed and unsafe (C, C++, Pascal, etc)
- It is strongly typed if it is typed and safe

—

Function calls: "Activation Record" is the layout for the required storage

Return address (where to put the return value). Dynamic link (top of the ARI of the caller). Parameters (values passed into function) Local variables

—

Stack smashing
A program writing more data to a block of memory than allocated for.
Buffer Overflow - Write data into adjacent memory locations (often the stack) by writing more data than the size of the buffer Overwriting control data - By overflowing a buffer, an attacker can overwrite control data like function return adress or instruction pointers Manipulating program flow - Manipulating control data can redirect to malicious code injected into the compromised buffer Execution of arbitrary code - Once the attacker controlse the program flow, they acn execute arbitrary code.
A good buffer overflow attack must overflow to the right place (the place to hold a return adress) Can be prevented by

- Bounds checking : Ensuring that buffers do not overflow by validating the size of data before copying it
- Safe string functions : Use secure string manipulation functions (strncpy instead of strcpy) that allow developers to specify the size of destination buffers (Similar to bounds checking)
- Address space randomization : Randomly arranging the memory addresses used by processes, making it harder to predict the location of data structure.
- Stack canaries : add random values between buffer and control data. If the canary is modified, it indicates a buffer overflow.
- Non-executable stack: Marking the stack region of memory as non-executable, preventing injected code from executing on the stack.

—
The biggest problem is stack smashing. C and C++ try to be highly efficient, but this means it does not check array bounds at runtime, etc.
Any code handling untrusted input is vulnerable:

- Code recieving input over an untrusted network
- Code accepting input from untrusted users on multi-user systems
- Code processing untrusted files
- Embedded software

Accepting input: gets(buf) - reads user input until the first end of line/file character. This is UNSAFE, since you may not encounter an end of line character, and the size of the input may be larger than than the size of the buffer.
fgets(buf, size, stdin) - reads user input until size-1 or EoL/EoF character.
strcpy(dest, src) - Copies string, making the assumption length(dest) >= length(src), and that src is null-terminated
strncpy(dest, src, size) checks for this.
C also implicitly casts between signed and unsigned integers, which can be problematic if the data read is not what is expected
Format strings: %p pops data from the stack, without checks. This is fine if there are exactly that many arguments passed into printf, but it can also pop other data. You can get around this by forcing it to be a string with `printf("%s", input)`
Solutions: Array bounds checking - High run-time overhead Non-executable memory - Program counter should point to the code segment, not the heap, stack, or data segments. Thes segments are non-executable, so an attacker cannot run injected code. Safe Library Compiler Solutions

- Canary: Low overhead, but this changes the layout of the stack frame of a function and is not compatible with some programs (debugger). It also only protects return addresses
  - Return Address Defender - Copy of the return address is saved somewhere secure.

Type-safe Language

- Enforce array bounds checking to stop buffer overflow

Static Source Code analysis

- Try to detect overflow vulnerabilities
- False positives and negative
- Limited understanding of runtime behaviour

Anomaly detection

- Detect data patterns that deviate from the norm

Code randomization
- Encrypt and decrypt instructions when they are prepared to be executed. This has a performance overhead though, and assumes that attacks are based on code injection

Memory Address Obfuscation/ASLR
- Randomizes the memory locations of process components (stack, heap and librarries) during runtime

—

ROP (Return Oriented Program) DEP: Data Execution Prevention Execute code, not data. Data areas are marked non-executable (Stack and Heap), and is hardware enforced EIP Control (Extended Instruction Pointer)
- EIP movement restricted: Can't jump to heap or stack
- Can only land in executable regions

These make it impossible to inject code directly: ROP lets you borrow pieces of code that alreay exist in executable regions of the process Then you create a sequence of operations to carry out an attack
Assume a series of cards where each card performs an operation to perform. This is available in the binary. Each card can be chained together to create complex operations. The cards are the Gadgets, and are already availabe in the process' executable regions

–

Ret2LibC technique Find Gadgets - Analyze the program's binary Controlled Stack - Use buffer overflow to overwrite the program's stack with controlled data, including the return adresses. Redirect control flow - towards the gadgets in LibC (instead of thes tart of shellcode, which may be blocked) Chain gadgets Execute ROP Chain Achieve Arbitrary Code Execution
You can set the return adress to point to a function in the LibC Prepare the facke stack frame with arguments for the function Return to the LibC function like "/bin/sh" to achieve desired outcome

—

You can try to overcome this using Behaviour Based Detection (Control Flow Integrity)
You want to build a control flow graph at compile time, and rewrite the binary to include IDs and ID checks. Then, you perform ID checks at run time (indirect jumps in control flow should have matching IDs). You want tomake sure the execution follows the path of the CFG
There are two graphs: Call graph and Control Flow Graph
Basic Block: Consecutive sequence of instructions with no jumps. Used for the Control Flow Graph
Call Graph: Nodes are functions, and there is an edge v_i, v_j if function v_i calles function v_j The CFG is defined by distinguishing calls and returns in the program. Each function is broken into basic blocks.
You use an in-line reference monitor
- Direct calls do not need to be monitored
- Indirect calls must be monitored because they are done through dynamic data on the stack

The monitor adds a label just before the target address of an indirect transfer When an indirect occurs, it checks if the label of the target is the same as the label of the possible destination If it isn't what we expect, we abort
Simplest Labeling
- Use same label for all targets
- It is insecure if an unintended location (malicious) also has label L

Detailed Labeling
- The return calls must all share the same label.

Assumptions: Unique IDs Non writable code Non executable Data
To defeat: Inject code with a legal label Modify code labels

—

Java Security
Components of Java Class Loaders Native Method Loader Heap Execution Engine JIT Compiler Security Manager

—

Java Security Models

"Sandbox" in which Java programs can run safely, without potential risk to systems or users

- Trusted code can read/write/delete any file, and start/kill threads and open/use/close socet connections without restrictions
  - Signed applets: containing a signature the browser can verify through a remotely running, independent certificate authority server

  POLICY file - configuration file used by the JRE to determine the granted permissions for each java program
- Untrusted code cannot access files, threads are hidden, and socket connections to the applet's orgin server are allowed

—

Main Componentse of Java Security Architecture

- Security Manager
  - Ensures the permissions specified in the policy file are not overridden
- Class Loaders
  - Seperate name spaces (classes loaded by a class loader instance belong to the same name space)
  - A class in one name space cannot access another name space
  - Classes from different web sites cannot access each other
  - Establish the set of permissions for a loaded class
  - Enforce a search order that prevents trusted system classes from being replaced by classes from less trusted sources
- Bytecode Verifier
  - Ensures variables are initialized before used
  - Rules for accessing private data and methods are not violated
  - Local variables accesses fall within the runtime stack (and does not overflow)
  - No "illegal" data conversions will occur
  - Parameters have appropriate types

—

Web Scripting Security Browser sends requests Web site sends response pages (which may include code) Network replies The interaction is subject to network attacks

The browser sends HTTP requests to the server (GET, POST, etc) The server replies with an HTTP response Stateless request/response protocol

A "cookie" is a name/value pair created by a website to store information in your computer (on your browser). It is used by web applications, and ONLY the website that created the cookie can read it.

Cross Site Scripting: Web pages can embed dynamic code that can be executed on the browser using script tags, alerts, events, etc. When the value (or processed value) of a text field is displayed on a website or stored then displayed, a hacker can introduce undesired commands into legitimate code that is executed by a browser on behalf of the web application.

The most common tags injected are

- `<script>` tags which reference either external JS code, or embed the code within the script tage itself
- `<body>` events (the event onload, attribute background, etc)

There are three types:

1. Reflected, where the script comes from the current HTTP request
   - When malicious JS is sent as part of the victim's request to the website
   - This is usually found in Phishing attacks
   - If successful, it can fuly compromise the user, since a hacker can view all the information that the user is able to view, modify any information that the user is able to modify
   - Initiate interactions with other application users, including malicious attacks
2. Stored XSS, where the malicious script comes from the website's database

- Hacker -> Server -> Request from user -> Malicious content is run on client (this may send data back to hacker)
- This malicious script is stored, and so is persistent. It is also automatically triggered when a user visits a page (regardless of how they get there), and affects all users. Higher risk

3. DOM-based XSS, where the vulnerability exists in client side code rather than server-side code.
   - DOM is a convention for representing objects in HTML/XML
   - It is the browser's internal representation of a webpage
   - In a DOM attack, the browser gives a script access to the DOM, so the script can change the pages's behaviour

A security policy for seperation among mutually untrusted scripts is Same Origin Policy. It isolates scripts and resources downloaded from different origins. Origin = domain name + protocol + port Same-origin policy applies to manipulating browser windows, manipulating frames, manipulating documents, manipulating cookies, URLs requested via the XmlHttpRequest

- Poorly enforced on some browsers
- Limitations if site hosts unrelated pages
- Can be bypassed in XSS attacks.

XSS attacks can be prevented by encoding/validation Blacklisting and whitelisting characters, or replacing characters

—

Functional Programming

eq? checks if two values are the same object in memory eqv? checks for equivalent in value and type equal? checks if two values are structurall equivalent, especilaly for complex data structures

`(define <identifier> <expression>)`

(display "string") # Display string (read-line) # read a string (read) # read an expression

‘ stops expressions from being evaluated. It can be used to write down a list

S-Expressions are a notation for nested list data

Lambda calculus - specifies the parameter and the mapping of a function in the following form Anonymous functions with immutability, and first-order functions The identity funtcion: $\lambda.(x) \mapsto x$ (lambda (arg1, arg2) procedure)

(cons) concatenates an element to the front of a list (car) takes the first element of a list or pair or (first) (cdr) takes the rest of the list or (rest) (append) concatenates lists together

Apply to all functions

(map function list) (filter function list) (foldl function starting list)

For, If

```
(for ([id sequence-expression])
  body ...
)
```

(in-range) generates a list with start, stop, step (optional) Iterate over nested lists with (in-list, or in-string) functions (like for each)

```
(for/list ([variable list])
  body
)

is the same as

(for ([i (in-list list)])
  body
)
(for* ([variable list] [variable list])
  body
)
```

nests clauses

`(if bool expr1 expr2)` expr2 is like the `else (when bool expr1)` more like a C-style if statement

—

Logic Programming
Programming paradigm based on principles of formal logic
Programs are written as assertions/rules, and a program execution involves making logical inferences to derive new information or reach conclusions
Facts: Statements about the world: likes(john, pizza). Rules: happy(X) :- likes(X, sunshine)
Queries: ?- likes(john, pizza).
Prolog uses recursion
Goal Statements:

- Statements or queries that the programmer wants to prove or satisfy -> the desired outcomes or the information sought

Axioms as a Theory:

- The collection of facts and rules for the "theory" of the logic program -> initial set of assumptions

Goal Statements as Theorems

- The theorems the programmer aims to prove from the theory

—
A predicate is a tuple: A relationship that defines a logical connection between its arguments N-ary relations:

- relations in Prolog can be n-ary -> involve more tan two entities

A predicate is a statement that defines a condition or property It has three parts: Name, Arguments (constants, variables, etc), Arity (number of arguments). It is basically each relationship happy() # (for example)
The arity is denoted with a /, i.e. loves/2 means that loves relates 2 things together
—
Variables can be used as both input and ouptup positions, and the prolog engine tries to bind variables in a way that satisfies the rules variables must start with uppercase letters or an underscore
—
Atoms are a basic data type represeting a constant (letters, numbers, or arbitrary symbols inside 'symbols')
—
:- "if/(implication)" , "and" ; "or" % "comment(multiline comments are C-style)" _ "placeholder/don't care" is "arithmetic expressions, i.e. X is 10 + 5" =:=, =\= arithmetic equality + inequality ==, \= succeeds if the two terms on either side cannot be unified ! stops backtracking one a match is found

- e.g. fact(N, 1) :- N=0,!.

\+ negation operator
—
Numbers can be expressed as integers or floating point numbers. They are sequences of digits, optionally preceded by a sign (+ or -) Floating point numbers are represeted at integer.fractionalpart (e exponent)
—
Rules allow for inferences (generally contain variables) employed(X) :- employs(Y, X)
—
Horn clauses H :- B1, B2, …, Bn, or H :- B1; B2; …; Bn CNF/DNF. If there are no body predicates (the rights side), the horn clause is a fact
— Prolog: Model: Facts & Rules, Queries Resolution (how to resolve a query by searching for rules and facts in order to prove a query true or false)

- If c1 and c2 are horn clauses, and the head of c2 matches one of the terms in the body of c1, then the term in c1 can be replaced with c2
- In other words, try to substitute each term in the body with a matching horn clause's body

Unification (the process of finding substitutions for variables in order to make terms identical)

- Variables that recieve a value during unification are called instantiated
- A constant unifies only with itself

- Two structures unify iff they share the same predicate and number of arguments, and the corresponding arguments unify recursively
- A variable can unify with anything
- [IMPORTANT] Two terms which can be unified are considered equal. The equality goal uses the infix predicate "="

Execution Order (DFS strategy. The order of clauses in the program can affect the behaviour of the program backtracking)

- Forward Chaining
  - Derives the goals from the axioms or facts in the knowledge base
  - Derives the goal from the axioms or facts in the knowledge base
- Backwards chaining
  - Starts with goals and attempts to resolve it by working backwards through rules and facts
  - More efficient in general
  - DFS backtracking

Backtracking (Allows exploring alternative paths when a particular search path fails to satisfy goal)

Lists, Cut, Negation, Arithmetic

—

Fuzz Testing

- A software testing technique that provides invalid, unexpected, or random input to a program to find vulnerabilities
- Complementary to traditional testing
- Test input generation: Random (generate input randomly -> broad coverage, but misses specific scenarios) vs Smart Fuzzing (involve more targeted and controlled input generation)
- Mutation Based Fuzzing: Modify valid inputs to create variations. Systematically alters input data to explore different execution paths, identify vulnerabilities
- Generation-based fuzzing: Create new inputs based on predefined specifications

Fuzz testing is often automated for efficient and systematic execution Monitor code coverage to track whcih parts of the program are exercised during testing This creates a feedback loop to identify vulnerabilities

It helps detect vulnerabilities early Preven exploitation Enhancing resilience Open source software security Comprehensive security testing Regulatory Compliance