DevSecCon | LONDON 20 OCT 2016

Static Analysis for Code and Infrastructure

By Nick Jones

MWR INFOSECURITY

Join the conversation #devseccon

The Problem

- Software developers make mistakes
- Mistakes = bugs = vulnerabilities
- Our goal is fewer bugs

Who Am I?

Nick Jones

- Security Consultant at MWR InfoSecurity
- Web application & infrastructure security
- Previous experience as a software developer

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control flow graphs
- Taint analysis
- Pointer tracking
- DevSecOps and static analysis

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control flow graphs
- Taint analysis
- Pointer tracking
- DevSecOps and static analysis

How Do We Find Bugs?

Static Analysis (SAST)

- Analysing an application without executing it
- Code review, binary analysis, reverse engineering

Dynamic Analysis (DAST)

- Analysing by monitoring and interacting with the application as it executes
- Fuzzing, tampering, functional testing

How Do We Find Bugs?

Static Analysis (SAST)

- Analysing an application without executing it
- Code review, binary analysis, reverse engineering

Dynamic Analysis (DAST)

- Analysing by monitoring and interacting with the application as it executes
- Fuzzing, tampering, functional testing

How Do We Find Bugs?

Static Analysis (SAST)

- Analysing an application without executing it
- Code review, binary analysis, reverse engineering

Dynamic Analysis (DAST)

- Analysing by monitoring and interacting with the application as it executes
- Fuzzing, tampering, functional testing

How Do We Code Review?

Manual

- Give code to smart security experts
- They read, understand and spot bugs

Automated

- Pass code to a tool
- Tool parses code, hunts for known issues

Code Review - Examples

```
void echo ()
{
        char buf[8];
        gets(buf);
        printf("%s\n", buf);
}
```

Code Review - Examples

```
webView.getSettings().setJavaScriptEnabled(true);
```

Manual Code Review – The Downsides

- Manual code review is expensive

~45 Million LOC

~86 Million LOC

~24 Million LOC

Manual Code Review – The Downsides

- Steve McConnell (Code Complete) says 10-20 defects per 1000 lines of code…

Manual Code Review – The Downsides

- Steve McConnell (Code Complete) says 10-20 defects per 1000 lines of code…

~675,000 bugs

~1,290,000 bugs

~360,000 bugs

Static Code Analysis

- Automated searching of source code for issues
- Higher up front costs
- ‘Free’ security once built and configured
- Catch low hanging fruit automatically

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control Flow Graphs
- Taint Analysis
- Pointer Tracking
- DevSecOps and Static Analysis

Computer Science Theory Ahead

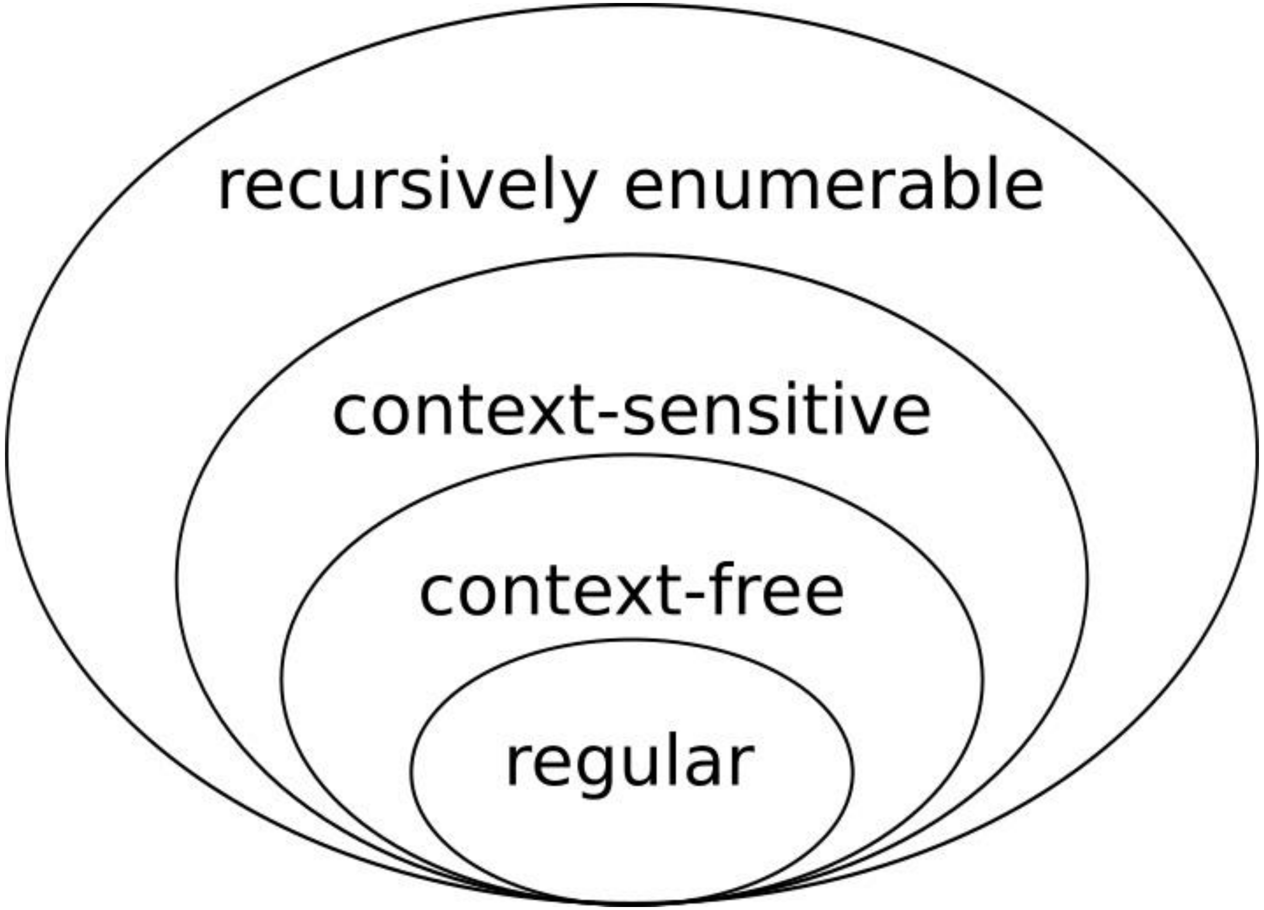
To best use tools, you need to understand them.

- Languages
- Automata / State Machines
- Parsers

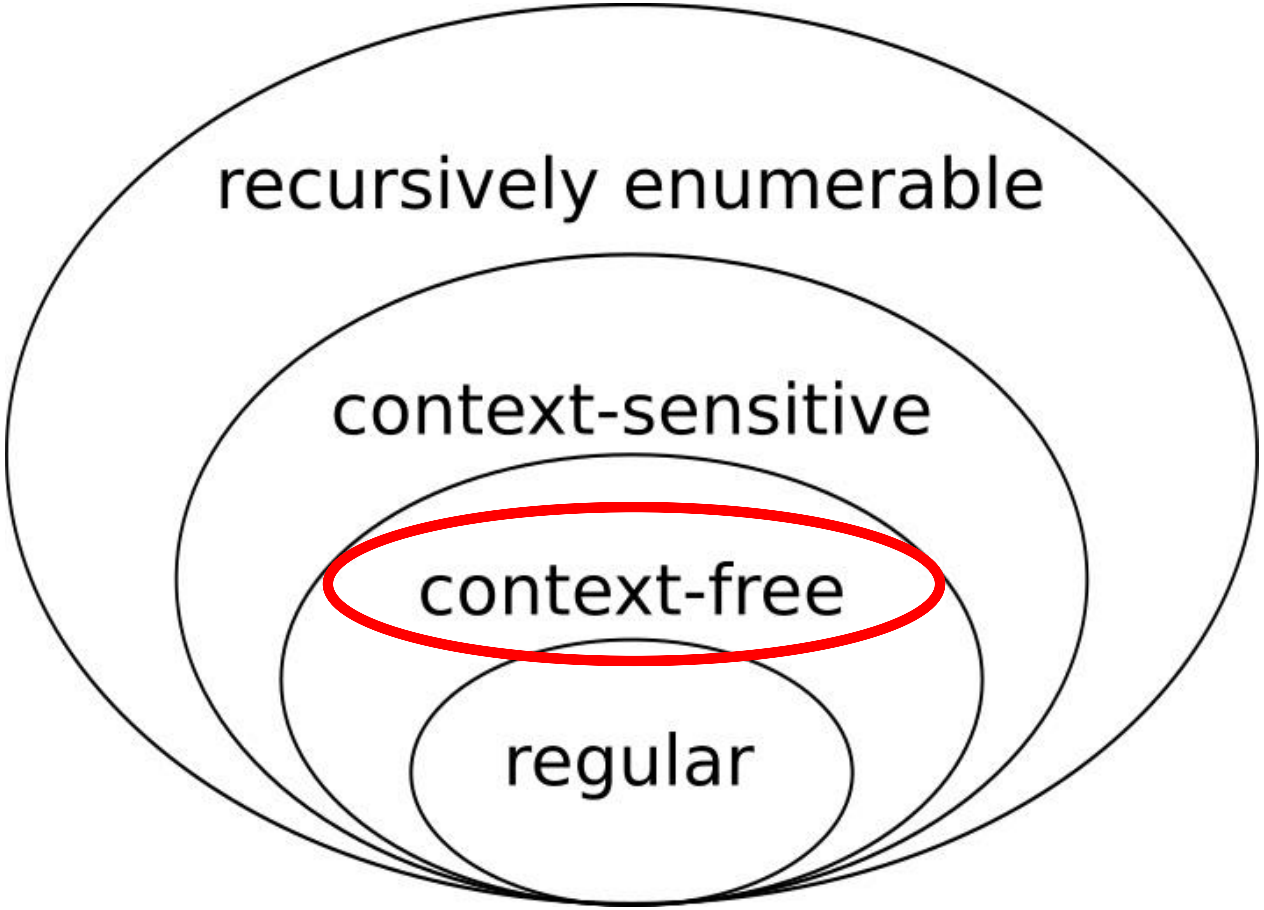
Languages

- Language - A set of strings of symbols constrained by a grammar
- Grammar – A set of rules defining the correct formation of a language
- Different grammars for different types of language

Chomsky’s Language Hierarchy



Chomsky's Language Hierarchy



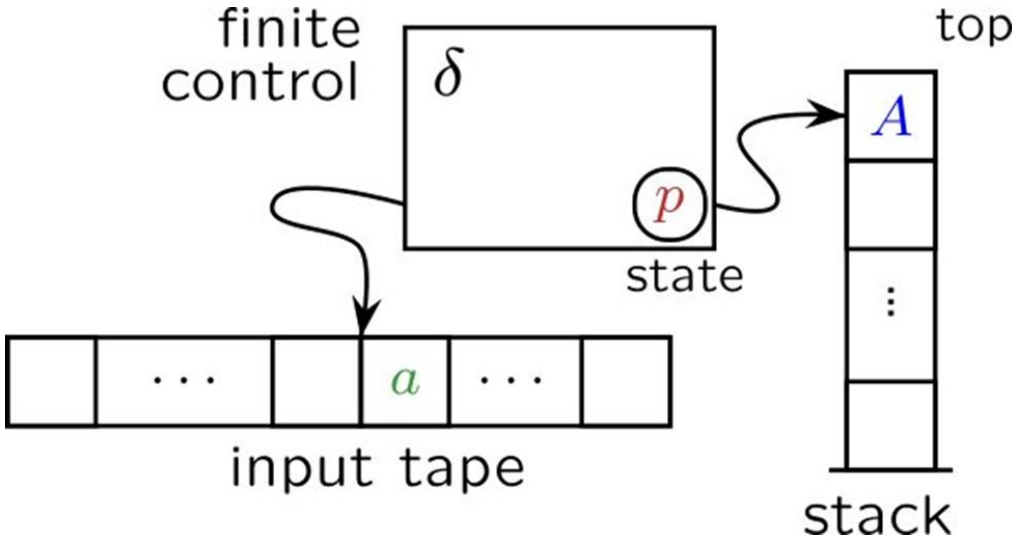
Context-Free Languages

- Anything that can be parsed by a context free grammar
- Most programming languages are mostly context free*
  - This is why parsing programming languages with regular expressions isn't great

* Templates, macros etc complicate this

Pushdown Automata

- Implementation of a context-free grammar
- Finite State Machines with stacks
- Decide transition based on both input and top of stack
- Can push/pop to stack as needed



Parsers

- Use a grammar to understand a language, convert it into a hierarchical data structure
- Several different types, depending on what you're parsing
- TL;DR: Construct a Parse Tree or Abstract Syntax Tree (AST) from the source code

Parsers

Two separate stages

- Lexer splits input text into tokens (strings with an understood meaning)
- Parser constructs AST or similar from list of tokens

Can combine both – scannerless parsing

Lexer Example

Code:

```
if (DEBUG)
{
      printf(…);
      printf(…);
      printf(…);
}
```

Lexer Example

Code:

```
if (DEBUG)
{
    printf(…);
    printf(…);
    printf(…);
}
```

Lexed Code:

```
if (DEBUG)
{
    printf(…);
    printf(…);
    printf(…);
}
```

Parser Example

Code:

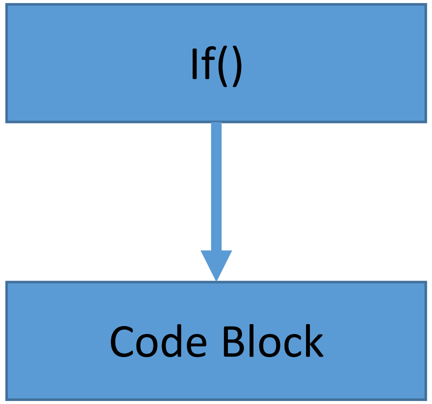
```
if (DEBUG)
{
     printf(…);
     printf(…);
     printf(…);
}
```

If()

Parser Example

Code:

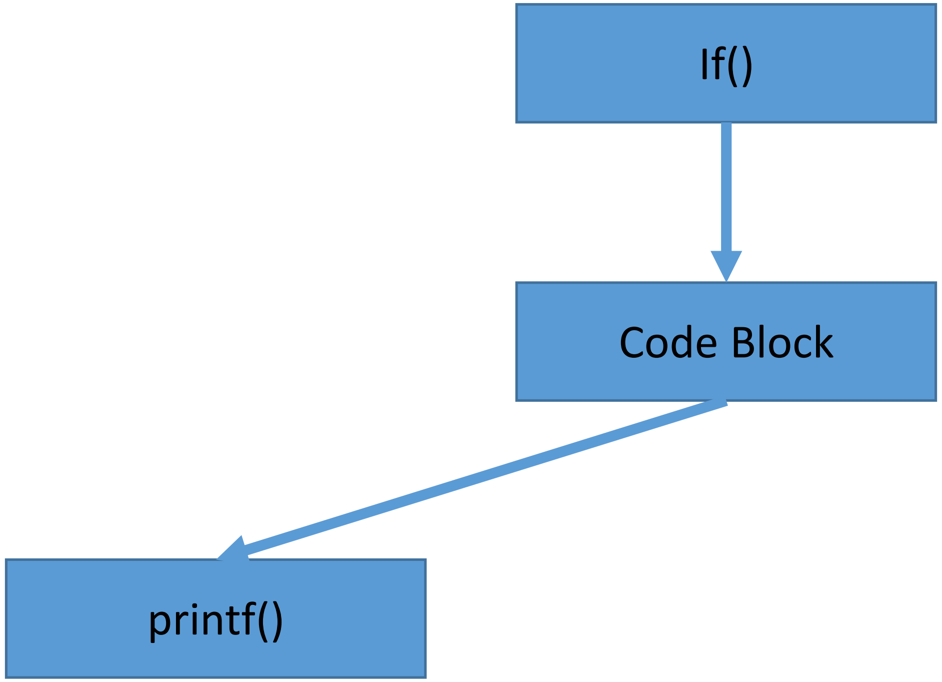
```
if (DEBUG)
{
      printf(…);
      printf(…);
      printf(…);
}
```



Parser Example

Code:

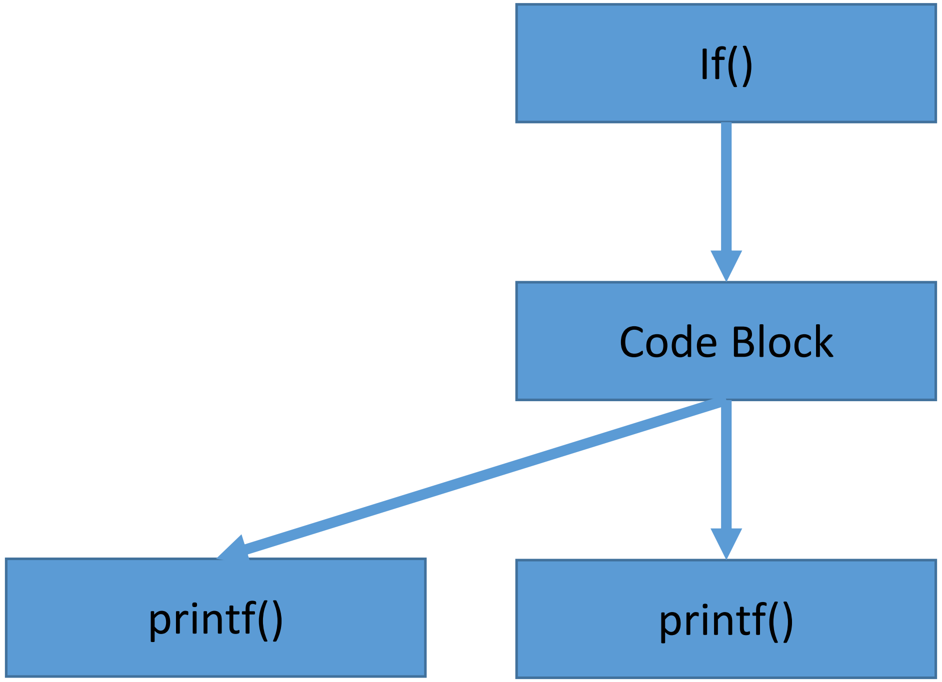
```
if (DEBUG)
{
      printf(…);
      printf(…);
      printf(…);
}
```



Parser Example

Code:

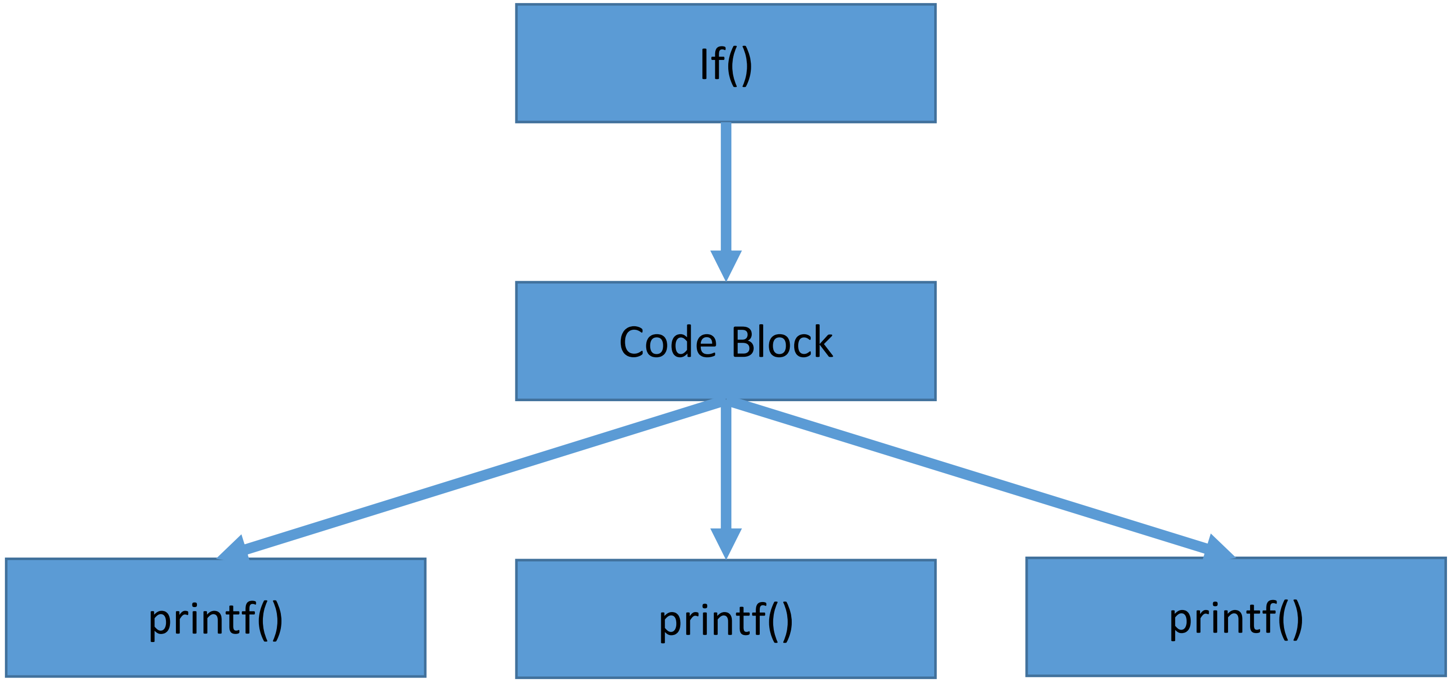
```
if (DEBUG)
{
      printf(…);
      printf(…);
      printf(…);
}
```



Parser Example

Code:

```
if (DEBUG)
{
      printf(…);
      printf(…);
      printf(…);
}
```



We’ve got an AST, now what?

Basic:

- Search AST for dodgy function calls, check for debug guards etc
- Check for questionable imports
- Can be done with regexes, but understanding of code structure -> fewer false positives

Advanced:

- Control Flow Graphs (CFGs)
- Taint Analysis

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control Flow Graphs
- Taint Analysis
- Pointer Tracking
- DevSecOps and Static Analysis

Control Flow Graphs

"a representation, using graph notation, of all paths that might be traversed through a program"

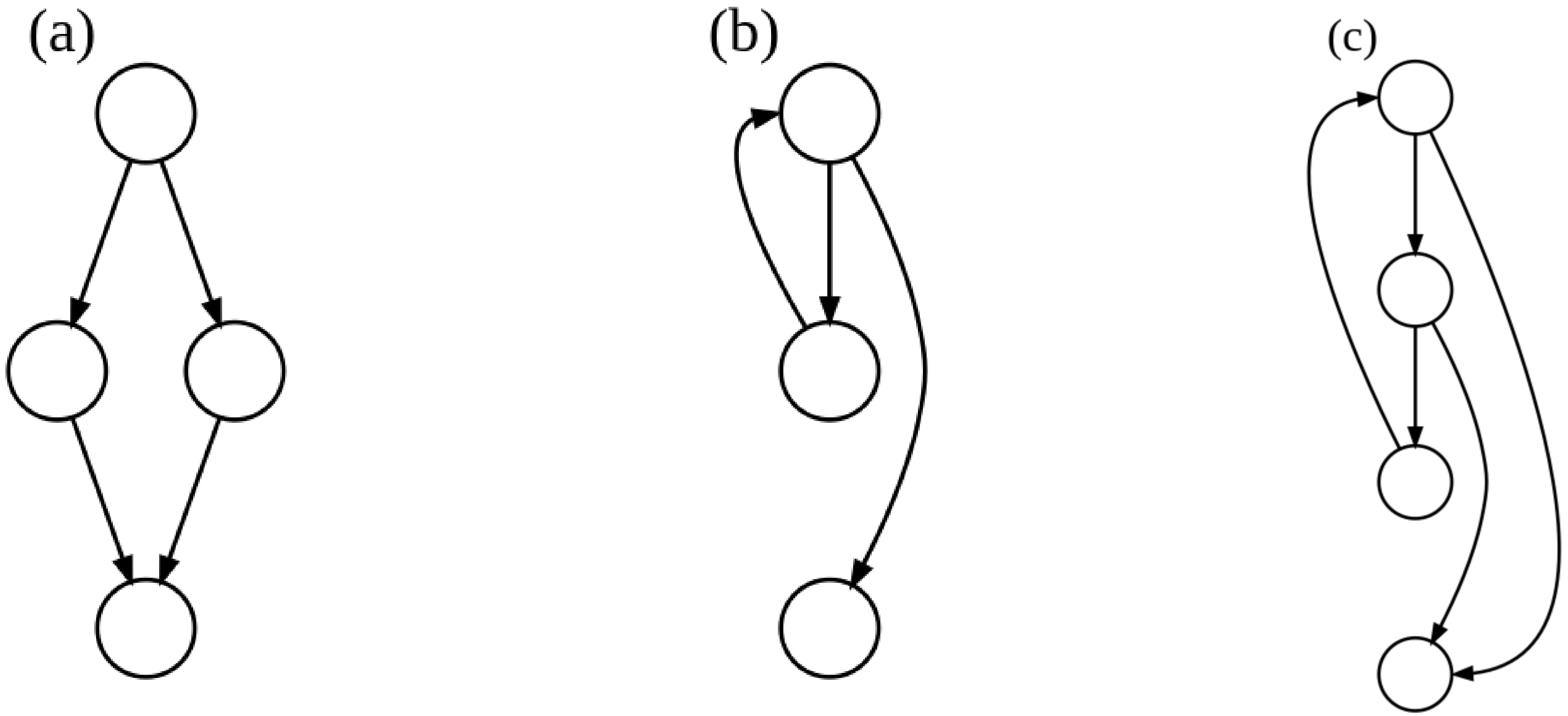
- Each basic block represented as a graph node
- Jump targets start block, jumps end block
- Jumps represented as directed edges

Control Flow Graphs

Commonly used for compiler optimisation

- Unreachable/dead code
- Detection of infinite loops
- Arithmetic optimisation
- Jump threading

Control Flow Graphs



Why Should I Care About Control Flow Graphs?

- Allows tracing of execution dependant on given inputs without running the application
- Allows a number of different analysis types
- We’re going to focus on:
  - Taint Analysis
  - Pointer Tracking

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control Flow Graphs
- Taint Analysis
- Pointer Tracking
- DevSecOps and Static Analysis

Taint Analysis

- Analyse data sinks to understand where the data has come from
- If it’s from external input, it’s tainted unless sanitised
- Trace data sinks back to original source
- Data sanitized several function calls ago? Trace the graph back and find it

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
    return login_query(user, password);
}

function login_query(user, password) {
    return mysqli_query('select * from user where user=' + 
$user + ' and password=' + $password + ';');
}
```

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
    return login_query(user, password);
}

function login_query(user, password) {
    return mysqli_query('select * from user where user=' +
$user + ' and password=' + $password + ';');
}
```

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
     return login_query(user, password);
}

function login_query(user, password) {
     return mysqli_query('select * from user where user=' +
$user + ' and password=' + $password + ';');
}
```

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
     return login_query(user, password);
}

function login_query(user, password) {
     return mysqli_query('select * from user where user=' +
$user + ' and password=' + $password + ';');
}
```

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
     return login_query(user, password);
}

function login_query(user, password) {
     return mysqli_query('select * from user where user=' +
$user + ' and password=' + $password + ';');
}
```

Taint Analysis

```
$result = login($_POST['user'], $_POST['password']);

function login(user, password) {
     return login_query(user, password);
}

function login_query(user, password) {
     return mysqli_query('select * from user where user=' +
$user + ' and password=' + $password + ';');
}
```

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control Flow Graphs
- Taint Analysis
- Pointer Tracking
- DevSecOps and Static Analysis

Pointer Tracking

- When walking the graph, track:
  - Pointer creation/destruction
  - Memory allocation/deallocation
- Spot code paths leading to memory errors

Pointer Tracking

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
        free(ptr);
}
...
if (DEBUG_MODE && err) {
    logError("operation aborted before commit", ptr);
}
```

Pointer Tracking

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
        free(ptr);
}
...
if (DEBUG_MODE && err) {
    logError("operation aborted before commit", ptr);
}
```

Pointer Tracking

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
        free(ptr);
}
...
if (DEBUG_MODE && err) {
    logError("operation aborted before commit", ptr);
}
```

Pointer Tracking

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
        free(ptr);
}
...
if (DEBUG_MODE && err) {
    logError("operation aborted before commit", ptr);
}
```

Pointer Tracking

Can be used to find:

- Null pointers
- Use after frees
- Dangling pointers

What Will We Be Covering?

- Why do we need static code analysis?
- How does an analyser work?
- Control Flow Graphs
- Taint Analysis
- Pointer Tracking
- DevSecOps and Static Analysis

Static Analysis in an SDLC

- Catch security issues before penetration tests
- One developer builds it, everyone can use it

- Can be built into existing toolchain, used with continuous integration systems etc.
- Catch issues as they are introduced to the codebase
- Catch regressions in code before it hits production

Static Analysis for Infrastructure

- Source code static analysis is known to work well
- Can we statically analyse infrastructure?

Infrastructure as Code

- Defining your infrastructure in software
- System definitions stored in configuration files, pushed/pulled to/from servers by agents or control nodes
- Common systems:
  - Chef
  - Puppet
  - Ansible
  - Salt

Infrastructure as Code

- Usually tested with unit and integration testing
  - Often as part of a CI toolchain
- Common tools:
  - BDD-Security
  - Cucumber
  - Rspec
  - Selenium

Static Analysis for Infrastructure

- Can we statically analyse infrastructure?
- Already common for syntax/style checks
  - Ansible -> ansible-lint
  - Chef -> FoodCritic, rubocop
  - Puppet -> puppet-lint, erb syntax checking
- Can be used to catch security issues too

What Security Issues Can We Find?

- Hardcoded passwords
  - Ansible -> ansible_become_pass without using ansible_vault or similar
- Presence of unnecessary tooling
  - gcc left on production servers
- Failure to apply hardening
  - SSH – password authentication/root login enabled
  - Overly permissive firewall rules
  - No SELinux/AppArmor/grsec

What Do We Need To Do This?

- Parser for CM tool's Domain Specific Language (DSL)
  - Most DSLs are variants on existing languages
  - Leverage existing parsers
- A rules engine
  - Define what "good" or "not good" looks like

- To analyse, walk the AST, compare tree nodes against rules DB
- Infrastructure static analysis simpler to implement yourself than code analysis

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Example

Rules:

```
Case lineinfile:
  if regexp.contains("PasswordAuthentication"):
    if line.matches("PasswordAuthentication yes"):
      raise_flag("PasswordAuthentication enabled on
        SSH")
Case ufw:
  if policy == "allow":
    raise_flag("UFW default incoming set to allow")
```

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Example

Tasks:

```
- name: Setup ufw
  ufw: state=enabled policy=deny
- name: allow password authentication
  lineinfile: dest=/etc/ssh/sshd_config
              regexp="^PasswordAuthentication"
              line="PasswordAuthentication yes"
              state=present
  notify: Restart ssh
```

Why Does This Help?

- Enforce common good practices in an environment agnostic manner
- Complements integration/unit testing
- Can be run locally on a developer’s machine
  - Instant feedback
  - No VMs required
- Complements automated integration testing as part of CI

Conclusion

- Static analysis catches some classes of bugs cheaply
- Build it into your continuous integration for automated security
- Static analysis can be used on IaC
  - Complements integration and unit testing

Thank you all for listening!

Any questions?