Lecture 16 - 12/11/2019

Container Security

Containers make achieving security much harder

- Shorter dev cycles
- More interactions
- Shared resources
- Public repositories are public
    - Anyone can put an image onto a public repo
- Container defaults can introduce vulnerabilities
    - E.g. “Expose 80”
- E.g. always specify a tag in the FROM statement
        - Don’t just use :latest

Achieving Security

Secure Images

- Everything that you use in an image can be a vulnerability.
- Only pull in the stuff that you actually need
- Don’t start with a base image from a public repo
    - Start from scratch if possible
- If you must start from a baes image, only use a base that has been signed.
    - And from a trusted source
    - Scan for vulnerabilities before using.

Secure Repositories

- Continuously monitor
- If possible, use a private repository
    - Access control
- Useful security features can be added to containers in that repo
    - You can add image metadata which is useful for tacking vulnerabilities
    - Tagging to filter or sort images
    - Automated policy checking

Secure deployment

- Ensure that all containers build on top of eachother
- Immutable container states

Secure Runtime

- Establish baseline behaviour for container in a normal, secure state.
- Network microservices: attach surface is large and complex
    - Allow only connectivity between container that actually need it
    - Restrict open ports and who can use them

Secure Orchestration

- Prevent risks from over-privileged accounts
- Prevent risks from attacks over the network
- Prevent risks from unwanted lateral movement.
- Configure orchestration to use proper access control
- Least privilege for each container
- White listing for specific containers

Secure the Host OS

- Scan for vulnerabilities
- Harden according to relevant guidelines/benchmarks
- Ensure container isolation

Continuously monitor for security

- Log every access to containers apps, services, systems, e.t.c.
- Performing regular audits of your log files
- Monitoring for anomalies
- Stay on top of current research

Information Flow

- Flow of information through a system
    - Confidentiality
        - Data of lower confidentiality flowing to a process of higher confidentiality
    - Integrity
        - Data of lower integrity flowing to a process of higher integrity
- This flow can happen two ways
    - Through code/programs
        - Compiler-based mechanisms to monitor and protect
        - Executable-based mechanisms (runtime mechanisms)
    - Through channels
        - System mechanisms to monitor and protect
        - Secure protocols to monitor and protect

Compiler based program protection

- Imagine code with two variables x and y
    - We can imagine a command sequence which might be several lines of code
    - Within this sequence it is defined that there is a flow of info from x to y if after this code we can look at y and infer something about x
- Explicit
    - y := x;
    - tmp := y; y := tmp;
- Implicit
    - If (x = 1) { y = 0; } else { y = 1; }
- If x is a variable, then x is the “Information flow class” of that variable
    - Info can flow from x to y, if $\underline{x} \le \underline{y}$ (confidentiality)
    - Info can flow from x to y, if $\underline{x} \ge \underline{y}$ (integrity)
- If there are several classes (e.g. $\underline{A}$, $\underline{B}$, $\underline{C}$)
    - I.e. least upper bound $\{\underline{A,B,C}\} \le \underline{y}$
- Compiler based protection mechanisms checks that info flows through a program are authorization
- A set of program statements is certified with respect to an info flow policy if the info flows in these statements do not violate policy.
- E.g. Consider the statements
    - If x = 1 then y := a;
else y := b
    - Information flows to *{x,a} into y*, or *{x,b} into y*.
    - If $\underline{a} \le \underline{y}$, $\underline{b} \le \underline{y}$ and $\underline{x} \le \underline{y}$ then the info flow is secure

- If the security depends on something such as time of day or some other unpredictable state, we can not verify with a compiler based mechanism.

Statements

- Assignment
- Compound
- Conditional
- Iterative
- Goto
- Proc
- Functions
- I/0 statements

E.g. y := f(x1,x2,x3 … xn)

Least upper bound {<u>x1</u>,<u>x2</u>,<u>x3</u> … <u>xn</u>} ≤ <u>y</u>

E.g.

```
If (x1 … xn) then
        S1;
Else
        S2;
```

Check for the information flows within S1 and S2 and
Glb = greatest lower bound
Lub {<u>x1</u> … <u>xn</u>} ≤ glb {<u>y</u> | y is the target of some assignment statement in s1 or s2}

E.g. infinite loop

```
Proc SupriseFlow (bool x, bool y) {
  y := 0;
  while x= 0 do
        (nothing)
  y := 1
}
```

X flows to y