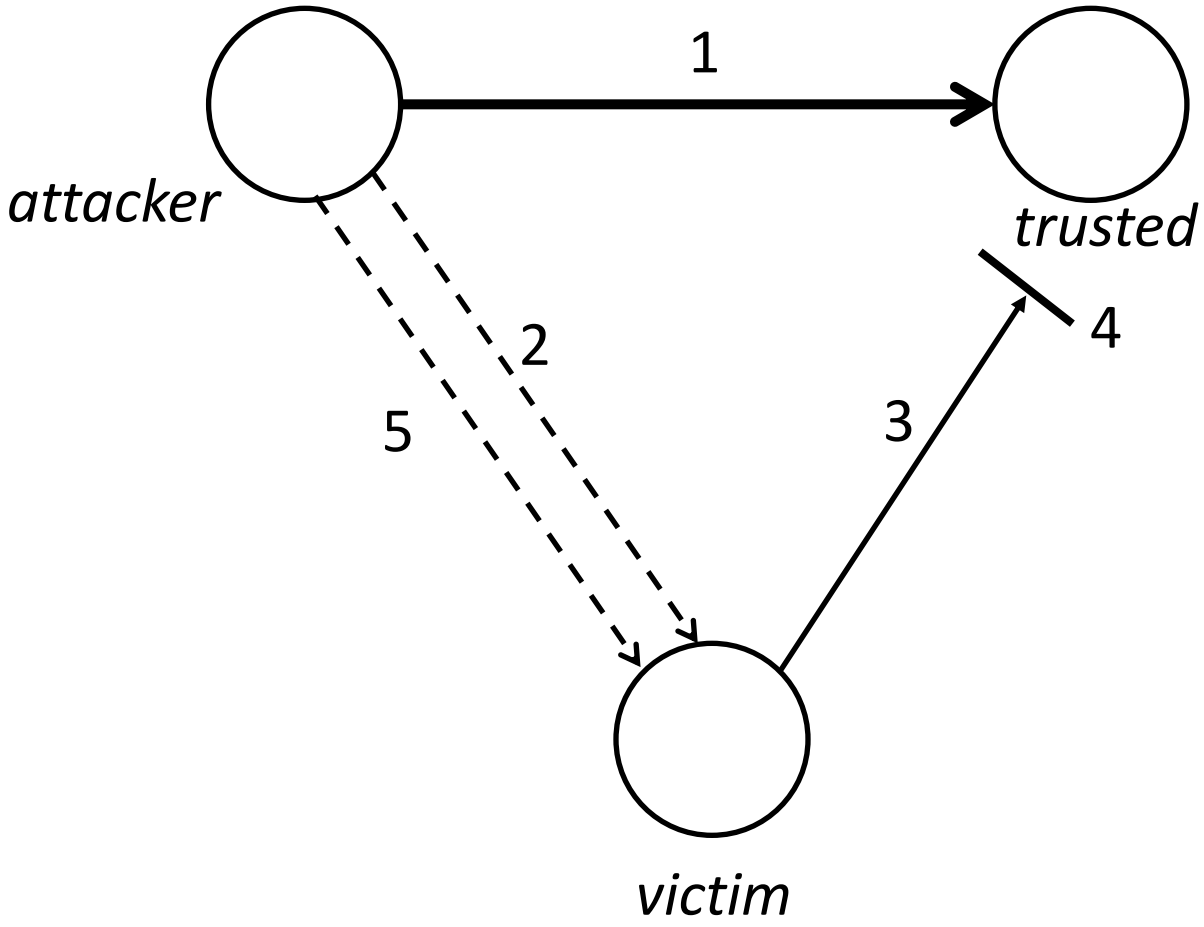
Lecture 23
November 20, 2024

Example: *rsh* Attack



1. *attacker* launches a DoS against *trusted*
2. *attacker* sends *victim* forged SYN, apparently from *trusted*
3. *victim* sends SYN/ACK to *trusted*
4. It never gets there due to DoS
5. *attacker* sends forged SYN/ACK to *trusted*, with command in data segment of packet
   - Need to know right sequence number
   - If so, causes command to be executed as though *trusted* requested it

Example: *rsh* Attack

- *Requires* capability: blocking of a connection between the *trusted* and *victim* hosts
  - Contains source address, destination address
  - Also time interval indicating when communication is blocked (ie, when the DoS attack is under way, and how long it lasts)
- *Provides* capability: execute command on *victim* host as if command were from *trusted* host
- *Concept*: spoof *trusted* host to *victim* host

JIGSAW Language

- Implements requires/provides model
- Capabilities: sets of typed attributes and values
  - **`extern`** keyword means it is defined elsewhere
- Concepts: two sets of capabilities
  - Required capabilities in **`requires`** block
  - Provided capabilities in **`provides`** block
  - **`action`** block lists actions to take when a concept is active

Example: JIGSAW Representation of *rsh* Attack

```
capability nosend is
      true_src, src, dst: type Host; # attacker, trusted, victim
      using: type Service;           # service to be exploited
end.
```

Structure of a capability:

- *using* is command to be executed, exploiting a service (here, *rsh*)

Example: JIGSAW Representation of *rsh* Attack

```
concept rsh_connection_spoofing is
        requires
                TP: type Trusted_Partner;          #- trusted host
                SA: type Active_Service;           #- service (here, rshd)
                PPS: type Prevent_Packet_Send;
                FPS: type Forged_Packet_Send;
                extern SNP: type SeqNumProbe;
```

PPS: capability for *true_src* to block *src* host receiving packets from *dst*

FPS: capability for *true_src* to send forget packet to *dst*

SNP: capability for *true_src* to determine next sequence number of *dst*

Example: JIGSAW Representation of *rsh* Attack

```
with          #- These instantiate the capabilities
      TP.service is RSH,          #- service is RSH
      PPS.host is TP.trusted,           #- blocked host = trusted host
      FPD.dst.host is TP.trustor,       #- spoofed packets go to host
                                        #-  trusting TP
      FPS.src is [PPS.host, PPS.port], #- apparent source of forged
                                        #-  packets is blocked
      SNP.dst is [SA.host, SA.port],  #- probed host must be
      SA.port is TCP/RSH,               #-  running RSH on usual port
      SA.service is RSH,
      SNP.dst is FPS.dst                #- forged packets go to probed
      active(FPS) during active(PPS)  #-  host while DoS of trusted
                                        #-  host is active
```

Example: JIGSAW Representation of *rsh* Attack

To meet **`requires`** conditions, relationships in **`with`** block must hold:

- Trusted host must be running *rsh* servicve
- Attacker must be able to block trusted host from sending packets to victim
- Attacker must be able to send spoofed packets ostensibly from trusted host to victim
- Attacker must know sequence number of packet victim sends to trusted host
- When attack on victim is being carried out, attack on trusted host must also be active

Example: JIGSAW Representation of *rsh* Attack

```
requires
        PSC: type push_channel;
        REX: type remote_execution;
```

PSC: capability to send code, commands to *dst*

REX: capability to execute that code, commands on *dst*

Example: JIGSAW Representation of *rsh* Attack

```
with          #- These set the new capabilities
      PSC.src <- FPS.true_src,          #- capability to move code from
      PSC.dst <- FPS.dst,               #-  attacker to rsh server
      PSC.true_src <- FPS.true_src,     #-  (victim)
      PSC.using <- rsh;
      REX.src <- FPS.true_src,          #- capability to execute code,
      REX.dst <- FPS.dst,               #-  commands on rsh server
      REX.true_src <- FPS.true_src,     #-  (victim)
      REX.using <- rsh;
end;
action
      true -> report("rsh connection spoofing: " + TP.hostname)
end;
```

Example: JIGSAW Representation of *rsh* Attack

- When all conditions in **`requires`** block satisfied, concept *rsh_connection_spoofing* is realized
- Attacker gets capabilities defined in **`provides`** section
  - Here, *PSC* and *REX* capabilities
- Events in **`action`** block executed
  - Here, message is printed to alert observer an *rsh* spoofing attack under way

Attack Graphs

- Describe attacks in terms of a general graph
  - Generalization of attack trees
- Used to represent attacks, detect attacks, guide penetration testing

Attack Graph and Penetration Testing

Here attack graph is a Petri net

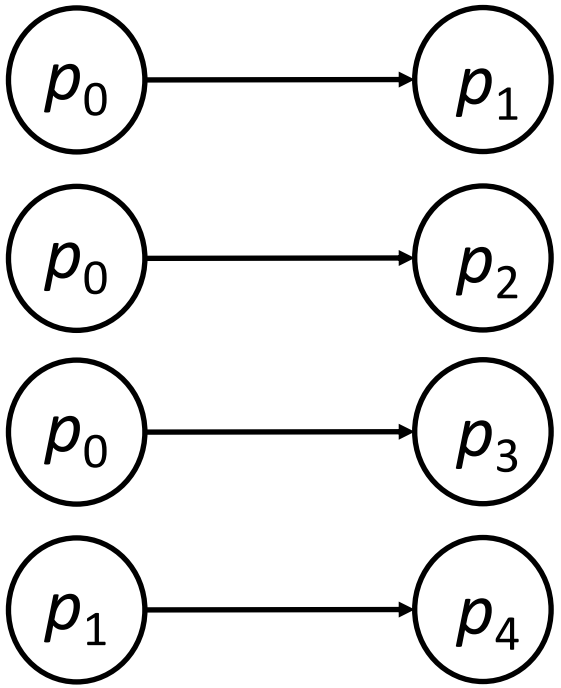
- Nodes $P = \{ p_1, ..., p_n \}$ states of entities relevant to system under attack
- Edges $T = \{ t_1, ..., t_m \}$ transitions between states
- Token on a node means attacker has appropriate control of that entity
- Tokens move to indicate progress of attack
- If node p_i precedes node p_j, attacker must get control of p_i before it can get control of p_j

Attack Graph and Penetration Testing

- McDermott: hypothesize individual flaws as 2 nodes connected by transition; then examine nodes for relationships that allow them to be linked

Attack Graph and *rsh* Attack

- First steps in attack:



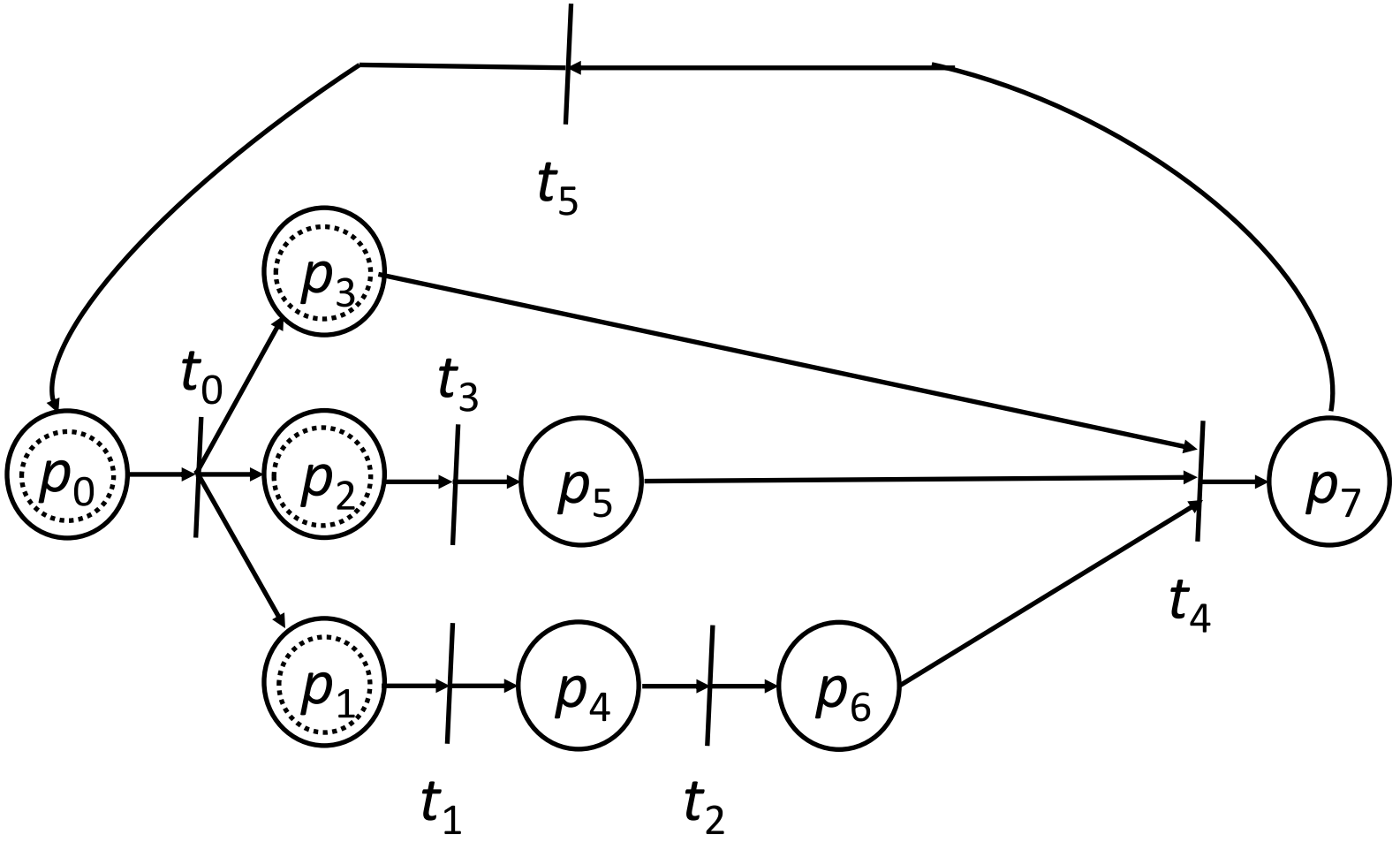
Initial scan of target

Identify an unused address

Establish that target trusts another host

Forge SYN packet

Attack Graph and *rsh* Attack



Petri net represents *rsh* attack

1. Before attack
2. After attack

Attack Graph and *rsh* Attack

States

- p_0: starting state
- p_1: found unused address on target network
- p_2: found trusted host
- p_3: found target that trusts the trusted host
- p_4: forged SYN packet created
- p_5: able to predict TCP sequence numbers of target host
- p_6: saturated state of network connections of trusted host
- p_7: final (compromised) state

Transitions

- t_0: attacker scanning system (splits into 3 transitions)
- t_1: attacker creating forged SYN packet
- t_2: attacker launching SYN flood against trusted host
- t_3: attacker figuring out how to predict victim's TCP sequence numbers
- t_4: forged SYN packet created
- t_5: attacker modifying trusted host file on victim
  - Attacker can now get *root* access on victim

Attack Graph and *rsh* Attack

- Attack starts at p_0
- t_0 splits into 3 transitions, as on success, 3 states of interest
- Need to instantiate all 3 states:
  - p_1: find unused address on target
  - p_2: find trusted host
  - p_3: find target that trusts trusted host
- t_1 is creating forged SYN packet
  - Transition from p_1 to p_4
- t_2 is attacker launching SYN flood (DoS) against trusted host
  - Transition from p_4 to p_6

Attack Graph and *rsh* Attack

- t_3: attacker figuring out how to predict victim's TCP sequence numbers
  - Transition from p_2 to p_4
- t_4: attacker launches attack using entities above
  - Transition from p_3, p_5, and p_6 to p_7
- t_5: attacker executes command
  - Example: modifying trusted hosts file to be able to get *root*

Intrusion Response

- Incident prevention
- Intrusion handling
  - Containment phase
  - Eradication phase
  - Follow-up phase
- Incident response groups

Incident Prevention

- Identify attack *before* it completes
- Prevent it from completing
- Jails useful for this
- IDS-based methods detect beginning of incidents and block their completion
- Diversity increases difficulty of attacks succeeding

Jailing

- Attacker placed in a confined environment that looks like a full, unrestricted environment
- Attacker may download files, but gets bogus ones
- Can imitate a slow system, or an unreliable one
- Useful to figure out what attacker wants
- MLS systems provide natural jails

Example Jail

- Cheswick recorded a break-in attempt using the SMTP server
- He created a very restrictive account, put the attacker in it
  - Monitored actions, including who the intruder was attacked
    - None succeeded and Cheswick notified the sysadmins of those systems
  - File system visible to attacker resembled UNIX file system
    - Lacked some programs that provided system information, or could reveal deception
    - Access times to critical files masked
- At request of management, finally shut down jail

Intrusion Detection System-Based Method

- Based on Intrusion Detection System (IDS) that monitored system calls
- IDS records anomalous system calls in locality frame buffer
  - When number of calls in buffer exceeded user-defined threshold, system delayed evaluation of system calls
  - If second threshold exceeded, process cannot spawn child
- Performance impact should be minimal on legitimate programs
  - System calls small part of runtime of most programs

Example Implementation

- Implemented in kernel of Linux system
- Test #1: *ssh* daemon
  - Detected attempt to use global password installed as back door in daemon
  - Connection slowed down significantly
  - When second threshold set to 1, attacker could not obtain login shell
- Test #2: *sendmail* daemon
  - Detected attempts to break in
  - Delays grew quickly to 2 hours per system call

Diversity

- Monoculture: an attack that works against one system works against all
- Diverse culture: one attack will not compromise all systems
  - Many different types of systems
  - Also can vary system configurations

Attack Surface and Moving Target Defense

- *Attack surface*: set of entry points, data that attackers can use to compromise system
- Usual approach: harden system to reduce attack surface, so more difficult for attackers to succeed
- *Defender's dilemma*: asymmetry between attacker, defender introduced by attack surface being non-empty
- *Moving target defense* (*MTD*): change attack surface while system runs
  - Attacks that work one time may not work another time
  - Reconnaissance data gathered as a prelude to attack no longer accurate after changes

Example: IP Address Hopping

- Client needs to contact server
- Component maps destination IP address, port number to different IP address, port number
  - These are chosen (pseudo)randomly
- When packet reaches network, another component remaps IP destination IP address, port number to real IP address, port number
  - If client, server on different networks, changed IP address must be on the same network as server
  - Mapping changes frequently (e.g., every minute)
- Attacker monitoring network cannot obtain real IP address, port number of server

Example: Mapping for Port Hopping

1. Divide time into discrete intervals of length τ at times $t_0, \ldots, t_i, \ldots$
   - At time k, port $p_k = f(k, s)$, where s is seed and f a pseudorandom number generator
   - Ports overlap at interval boundaries
   - So if L amount of overlap, p_k valid over interval $[t_k - L_\tau, t_k + L_\tau]$
2. Use encryption algorithm for mapping
   - Low-order octet of IP address and port number enciphered
   - High octet of result is low-order octet of IP address, rest is port number
   - Remapping just reverses encryption to get real IP address, port number

Notes on Moving Target Defenses

- Network-based MTDs
  - Must rely on randomness to prevent attacker from predicting changes to attack surface
  - Defender must distinguish between clients authorized to connect and clients not authorized to connect
- Host-based MTDs
  - Also must rely on randomness to prevent attacker from predicting changes to attack surface
  - Here, attacker is typically authorized to have access to some account in some way
  - Attack surface is within host

Address Space Layout Randomization

- Executables have several segments
  - Exact number, layout depends on compiler and systems
- When loaded into memory, segments arranged in particular order
  - That way, positions of variables, functions fixed in virtual memory
  - Attack tools exploit knowing where these are
- *Address space layout randomization* (ASLR) perturb the placement of segments, variables, functions
  - Then attack tools exploiting knowing where segments, variables, functions won't work

Address Space Layout Randomization

- Key question: how is perturbation done?
- Simplest: randomize placement of segments in virtual memory
- Others
  - Randomize order and/or locations of variables, functions within segments
  - Add random amount of space between variables, between functions
- Effectiveness depends on entropy introduced into address space
  - 32-bit Linux: uncertainty of segment base typically 16 bits, so easy to use brute force attack
  - 64-bit Linux: uncertainty of segment base typically 40 bits, so a search takes long enough that it is likely to be detected

Intrusion Handling

- Restoring system to satisfying site security policy
- Six phases
  - *Preparation* for attack (before attack detected)
  - *Identification* of attack
  - *Containment* of attack (confinement)
  - *Eradication* of attack (stop attack)
  - *Recovery* from attack (restore system to secure state)
  - *Follow-up* to attack (analysis and other actions)
- Discussed in what follows

Containment Phase

- Goal: limit access of attacker to system resources
- Two methods
  - Passive monitoring
  - Constraining access

Passive Monitoring

- Records attacker's actions; does *not* interfere with attack
  - Idea is to find out what the attacker is after and/or methods the attacker is using
- Problem: attacked system is vulnerable throughout
  - Attacker can also attack other systems
- Example: type of operating system can be derived from settings of TCP and IP packets of incoming connections
  - Analyst draws conclusions about source of attack
  - *nmap* does this; usually successful

Constraining Actions

- Reduce protection domain of attacker
- Problem: if defenders do not know what attacker is after, reduced protection domain may contain what the attacker is after
  - Stoll created document that attacker downloaded
  - Download took several hours, during which the phone call was traced to Germany

Example: Honeypots

- Entities designed to entice attacker to do something
- *Honeyfiles*, *honeydocuments*: designed to entice attackers to read or download it
  - Stoll used this to keep intruder on line long enough to be traced (internationally)
- *Honeypots*, *decoy servers*: servers offering many targets for attackers
  - Idea is attackers will take actions on them that reveal goals
  - These are instrumented, monitored closely
- *Honeynets*: like honeypots, but a full network
  - Treated like honeypots

Deception

- Cohen's Deception Tool Kit
  - Creates false network interface
  - Can present any network configuration to attackers
  - When probed, can return wide range of vulnerabilities
  - Attacker wastes time attacking non-existent systems while analyst collects and analyzes attacks to determine goals and abilities of attacker
  - Experiments showed deception is effective response to keep attackers from targeting real systems

Example: Honeynet Project

- International project created to learn about attacker community
- Phase 1: identify common threats against specific OSes, configurations
  - Gen-I honeypots crude but very effective
- Phase 2: collect data more efficiently
  - Gen-II honeypots easier to deploy and harder to detect
- Used to gather attack signatures, enable defenders to handle attacks without endangering production systems

Eradication Phase

- Usual approach: deny or remove access to system, or terminate processes involved in attack
- Use wrappers to implement access control
  - Example: wrap system calls
    - On invocation, wrapper takes control of process
    - Wrapper can log call, deny access, do intrusion detection
    - Experiments focusing on intrusion detection used multiple wrappers to terminate suspicious processes
  - Example: network connections
    - Wrapper around servers log, do access control on, incoming connections and control access to Web-based databases