What is Moving Target Defense (MTD)?

- Aim to substantially increase the cost of attacks by deploying and operating networks/systems to make them less deterministic, less homogeneous, and less static.
- Continually shift and change over time to increase complexity and cost for attackers, limit the exposure of vulnerabilities and opportunities for attack, and increase system resiliency.
- Dynamically altered in ways that are manageable by the defender yet make the attack space appear unpredictable to the attacker.

http://cybersecurity.nitrd.gov/page/moving-target

What is Moving Target Defense (MTD)?

- Also known as “Cyber Maneuver”, “Adaptive Cyber Defense”
  - Reactive ➔ Proactive
  - Static ➔ dynamic
- Enables defenders to create, analyze, evaluate, and deploy mechanisms and strategies that are
  - continually shift and change over time to increase complexity and cost for attackers
  - limit the exposure of vulnerabilities and opportunities for attack, and increase system resiliency.

Against the expert in defense, the enemy does not know where to attack. —SunTzu

Cyber Kill Chain

- Put forth by Lockheed Martin to describe phases of a targeted attack
  1. Reconnaissance: The attacker collects useful information about the target.
  2. Access: The attacker tries to connect or communicate with the target to identify its properties (versions, vulnerabilities, configurations, etc.).
  3. Exploit Development: The attacker develops an exploit for a vulnerability in the system in order to gain a foothold or escalate his privilege.
  4. Attack Launch: The attacker delivers the exploit to the target. This can be through a network connection, using phishing-like attacks, or using a more sophisticated supply chain or gap jumping attack (e.g., infected USB drive).
  5. Persistence: The attacker installs additional backdoors or access channels to keep his persistence access to the system.

MTD Categories

- System-based MTD
  - Software-based
    - Application, OS, Data
  - Hardware-based: processor, FPGA, platform
- Network-based MTD
  - MAC layer: changing MAC address
  - IP layer: IP randomization
  - TCP (Traffic) layer: changing network protocol
  - Session layer
Software-based MTD

- Goals
  - Prevent unwanted modification
  - Protect software against analysis

- Types
  1. Dynamic Runtime Environment: Address Space Layout Randomization (ASLR), Instruction Set Randomization,
  2. Dynamic software: In-place code randomization, Compiler-based Software Diversity
  3. Dynamic Data

Network-based MTD

- Network reconnaissance is the first step for attackers to collect network and host information and prepare for future targeted attacks.

- Goal: make the scanning results expire soon or give the attacker a different view of the target system

- Examples: IP randomization, Port randomization

How to analyze one MTD?

- Specific threat model
- Technical details
- Cost and benefit
- Weakness and Improvements

Threat Model

- Data leakage attacks, e.g., steal crypto keys from memory
- Denial of Service attacks, i.e., exhaust or manipulate resources in the systems
- Injection attacks
  - Code injection: buffer overflow, ROP, SQL injection
  - Control injection: return-oriented programming (ROP)
- Spoofing attack, e.g., man-in-the-middle
- Authentication exploitation: cross-site scripting (XSS)
- Scanning, e.g., port scanning
- Physical attack: malicious processor

Outline

- Introduction of Moving Target Defense
- How to defeat Buffer Overflow attacks?
  - Address Space Layout Randomization
- How to defeat port scanning attacks?
  - IP/port randomization

Threat Model

- Data leakage attacks, e.g., steal crypto keys from memory
- Denial of Service attacks, i.e., exhaust or manipulate resources in the systems
- Injection attacks
  - Code injection: buffer overflow, return-oriented programming (ROP)
  - Control injection: return-oriented programming (ROP)
- Spoofing attack, e.g., man-in-the-middle
- Authentication exploitation: cross-site scripting (XSS)
- Scanning, e.g., port scanning
- Physical attack: malicious processor
Buffer Overflow Attacks
- Also known as Buffer overrun, BOF.
- First major exploit: 1988 Internet Worm, Robert Morris.
  - May exploit buffer overflow in finger service.
  - 26 years old techniques
- Heartbleed attack, 2014
  - Due to implementation bug on OpenSSL library → Fail to check the length of Heartbeat request message
  - Leaking encryption key and user/password
  - Easy to fix, but might have been used as a zero-day attack for at least two years.

Computer Buffer
- Buffer: A contiguous block of computer memory, can be used for
  - Data: variables (static/global, dynamic/local), arrays
  - Code: user programs, shared libraries, kernel programs.
- To shield User/kernel programs from each other, virtual memory is used
- Within a virtual memory address space, different OS/CPUs have different ways to allocate buffers.
- On Linux, static/global variables allocated at load time on the data segment, dynamic/local variables are allocated at run time on the stack.

Segment Layout of Linux Process

What are BOF attacks?
- They attack corrupt data values in memory adjacent to a buffer by writing outside its bounds
- Stack-based exploitation
  - Discover vulnerable code
  - Overwrite the return address
  - New return address points to alternate code
  - Inject shellcode into the stack or use existing code (return-oriented programming, ROP)
- Heap-based exploitation
  - Insert instructions into the heap and then trick the program into executing them.

Stack Frame

Stack Overflow Example
- Suppose a web server contains a function:
  ```
  char a[30];
  void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
  }
  ```
- When the function is invoked the stack looks like:
  - Suppose `str` is 136 bytes long, after `strcpy`:
    - Buffer overflow:
      - `buf` is overwritten with `str`
      - `buf` points to a new location
      - New return address points to alternate code
      - Injection of shellcode or existing code
    - Return-oriented programming (ROP)

Basic Stack Exploit

- Main problem: no range checking in `strcpy()`.
- Suppose *str is such that after `strcpy` stack looks like:

```
<table>
<thead>
<tr>
<th>Code for P</th>
<th>Program P: <code>exec( &quot;/bin/sh&quot; )</code></th>
</tr>
</thead>
</table>
```

- When `func()` exits, the user will be given a shell.
- Note: attack code runs in stack.
- To determine ret guess position of stack when `func()` is called.

BOF Mitigations

- Proper programming language application
- Safe library usage
- Executable Space Protection
- Address Space Layout Randomization (ASLR)
- Deep Packet Inspection (DPI)
- Pointer Protection

Problem: Lack of Diversity

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to hijack control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine
  - Prevents attackers from using the same exploit code against all instantiations of the same program.

Address Space Layout Randomization (ASLR)

- Randomly choose base address of stack, heap, code segment
- Randomly pad stack frames and malloc() calls
- Randomize location of Global Offset Table
- Randomization can be done at compile- or link-time, or by rewriting existing binaries
  - Threat: attack repeatedly probes randomized binary

Segment Layout of Linux Process with PaX

- Linux kernel patch
- Goal: prevent execution of arbitrary code in an existing process’s memory space
- Enable executable/non-executable memory pages
- Any section not marked as executable in ELF binary is non-executable by default
  - Stack, heap, anonymous memory regions
- Access control in `mmap()`, `mprotect()` prevents unsafe changes to protection state at runtime
- Randomize address space layout

PaX

- http://pax.grsecurity.net

**PaX ASLR**

- PaX applies ASLR to ELF binaries and dynamic libraries.
- User address space consists of three areas
  - Executable, mapped, stack
- Base of each area shifted by a random "delta" (on x86)
  - Executable: 16-bit random shift
    - Program code, uninitialized data, initialized data
  - Mapped: 16-bit random shift
    - Heap, dynamic libraries, thread stacks, shared memory
  - Stack: 24-bit random shift
    - Main user stack

**PaX RANDUSTACK**

- Responsible for randomizing userspace stack
- Userspace stack is created by the kernel upon each execve() system call
  - Allocates appropriate number of pages
  - Maps pages to process’s virtual address space
    - Userspace stack is usually mapped at 0xBFFFFFFF, but PaX chooses a random base address
  - In addition to base address, PaX randomizes the range of allocated memory

**PaX RANDKSTACK**

- Linux assigns two pages of kernel memory for each process to be used during the execution of system calls, interrupts, and exceptions
- PaX randomizes each process’s kernel stack pointer before returning from kernel to userspace
  - 5 bits of randomness
- Each system call is randomized differently
  - By contrast, user stack is randomized once when the user process is invoked for the first time

**PaX RANDMMAP**

- Linux heap allocation: do_mmap() starts at the base of the process’s unmapped memory and looks for the first unallocated chunk which is large enough
- PaX: add a random delta_mmap to the base address before looking for new memory
  - 16 bits of randomness

**PaX RANDEXEC**

- Randomizes location of ELF binaries in memory
- Problem if the binary was created by a linker which assumed that it will be loaded at a fixed address and omitted relocation information
  - PaX maps the binary to its normal location, but makes it non-executable + creates an executable mirror copy at a random location
  - Access to the normal location produces a page fault
  - Page handler redirects to the mirror “if safe”
  - Looks for “signatures” of return-to-libc attacks and may result in false positives

**Base-Address Randomization**

- Only the base address is randomized
  - Layouts of stack and library table remain the same
  - Relative distances between memory objects are not changed by base address randomization
- To attack, it’s enough to guess the base shift
  - A 16-bit value can be guessed by brute force
    - 2^16 (on average) overflows with different values for addr of known library function – how long does it take?
      - Shacham et al. attacked Apache with return-to-libc
      - 216 seconds
    - If address is wrong, target will simply crash
Summary

• Randomness is a potential defense mechanism
• Many issues for proper implementation
• Serious limitations on 32-bit architecture
  – On 32-bit systems, runtime randomization cannot provide more than 16-20 bits of entropy
• How about being combined with “a crash detection and reaction mechanism” called watcher?
  – May not respond quick enough
  – May suffer from DoS attack

Improvements

• Move to 64-bit architecture
  – At least 40 bits of randomization
  – Entropy is high enough, and easy to detect attacks of this magnitude.
• Frequent Re-randomization
  – Randomize the address space layout of a process more frequently after process creation.
  – Adds no more than 1 bit of security against brute force attacks regardless of the frequency, $2^{n-1}$ vs. $2^n$
  – It can mitigate the damage when the layout of a fixed randomized address space is leaked through other channels

Outline

• Introduction of Moving Target Defense
• How to defeat Buffer Overflow attacks?
  – Address Space Layout Randomization
• How to defeat port scanning attacks?
  – IP/port randomization

Threat Model

• Data leakage attacks, e.g., steal crypto keys from memory
• Denial of Service attacks, i.e., exhaust or manipulate resources in the systems
• Injection attacks
  – Code injection: buffer overflow, SQL injection
  – Control injection: return-oriented programming (ROP)
• Spoofing attack, e.g., man-in-the-middle
• Authentication exploitation: cross-site scripting (XSS)
• Scanning, e.g., port scanning, IP scanning for targeted attack
• Physical attack: malicious processor

Dynamic Virtualized Network Topology

• 3-level decoys
  – VM level: KVM
  – OS level: OpenVZ/LXC
  – Process level: Honeyd
• Dynamic Network Topology
  – Centralized controller in the hypervisor
Two Challenges in Network-based MTD

1. Service availability
   - Authenticated clients should always know the new IP address/port number.
   - When the IP and Port changes, the connection still maintained, minimizing service downtime.

2. Service Security
   - Only the authenticated users can access the service.
   - How to mitigate insider attacks?

Live VM Migration

- VMware vMotion
  - Three execution states for moving a VM without service interruption
  - Disk state: shared storage such as SAN and NAS
  - Memory state: trace phase $\Rightarrow$ pre-copy phase $\Rightarrow$ switchover phase
  - Network State: Virtual switch, virtual NIC
    - Require source and destination hosts on the same subnet.
    - Migration should be fast to prevent network connection timeout.

- Our solution in MTD
  - Currently, no need to transmit the disk and memory states
  - Source and destination hosts can be on different subnets.
  - Allows longer connection timeout for migration

Seamless TCP Connection Migration

- Keep end-to-end transport connection alive through separating transport endpoint identification from network endpoint identification.

- Three components
  - Connection virtualization
  - Connection translation
  - Connection migration

Connection Virtualization

- Internal address for applications;
  - IP and Ports
  - never changes for one connection
- External address for communications
  - IP and Ports
  - may change according to MTD requirements
- A map to translate between Internal address and External address

Connection Translation

At beginning, internal address $\Rightarrow$ external addresses

Server changes its IP address
Network Migration

- After the server changes its IP address and port, it will inform the client to update the internal-external address mapping.
- Migration Steps: protected by a shared secret key
  - Suspend a connection
    - Keep connection alive
  - Resume a connection
    - Update internal-external endpoints mappings
    - Server sends UPDATE packet
    - Client sends UPDATE_ACK packet
  - Both endpoints need to know the same internal address pair.

Implementation

- All in a kernel module in Linux
- Support both client and server mobility
- Connection Virtualization
  - Intercept socket system calls
- Connection Translation
  - Instrument Netfilter hooks
- Connection Migration
  - Migration daemon

Intercept System Calls

- Overwrite the function pointers in the system call table
- Intercept
  - Accept()
  - Connect()
  - Close()
  - Getsockname()
  - Getpeername()

Instrument Netfilter hooks

- For outgoing traffic
  - NF_IP_LOCAL_OUT for destination address translation
  - NF_IP_POSTROUTING for source address translation
- For incoming traffic
  - NF_IP_PREROUTING for destination address translation
  - NF_IP_LOCAL_IN for source address translation

Migration Daemon

- A Kernel thread as a server process
- Initiate the suspension after receiving a suspend event from APM
  - Active the connection migration helper
- Restore the connection after receiving a resumption event from APM
  - Exchange UPDATE and UPDATE_ACK packets to update the internal to external address mapping

Cost and Limitation

- Require a large number of decoys (fake node)
  - Memory, CPU, network overhead
  - High-interaction vs. low-interaction decoys
- Cannot prevent insider attacks
References

- www.cs.utexas.edu/~shmat/courses/cs338b_fall09/04aslr.ppt