**Mem Allocation on the stack (from high address to low): 1.** Stack(storage for called function and return data) **2.** heap(storage for dynamically created data) **3.** static data(global variables) **4.** Code/Text(instructions of the program)

esp esp esp **EX:** void func(int x) {int a=0; int b=x;}

| | 10 | 0 | mains EBP| return add| 10 | int main() {func(10);}

low add ebp high address ESP keeps getting moved to low add, EBP points to top of stack (higher add)

**VULNERABILITY BUFFER OVERFLOW:** more data is put in buffer(memory block) than it can hold **ATTACK:** steal private info, corrupt valuable info, execute malicious code (1.Load assembly code into mem 2.Get instruction pointer to point to it(find correct return address)) **EXAMPLES**: strcpy operates on null terminated strings, no bound checks, data overwritten in adjacent memory, SEGV because address is outside of the process address space, skip instructions, injected data causes crash, security variable is overwritten. // **COUNTERMEASURES:**stay within bounds/check lengths before writing/limit input to certain characters/limit programs privileges/canary values// 0x11223344->Big endian(11223344-big end first)/little endian(44332211-little end first): different ways to store in mem **VULNERABILITY: OFF-BY-ONE:** miscalculating condition to end a loop(ie: i<=10 vs i<10), **NULL Termination**=null byte written beyond end of buffer, some strings always place a NULL terminator at the end of a string, single byte of mem overwritten(ie: strncat()) **ATTACK:** overwrite EBP to execute malicious code,stack frames location of caller function changed which means local var and return address changes SO you can write a bogus stack frame, execute arbitrary code after function returns. **VULNERABILITY: INTEGER/ARITHMETIC:** miscalculations of buffer size, occur when an operation on an int value causes it to increase/decrease past what it should. Also related to **SIGNEDNESS BUGS**, usually results in "wrap-around" value **VULNERABILITY: FORMAT STRING:** %n is most dangerous-causes number of bytes output so far to be written to address of pointer passed in as the associated parameter **ATTACKS:** Crashing the system=printf ("%s%s%s…"); Viewing the stack=printf ("%08x %08x %08x %08x %08x\n"); Viewing memory at a location as specified=printf ("\x10\x01\x48\x08 %x %x %x %x %s"); Writing an integer to a location in the process memory printf ("\x10\x01\x48\x08 %x %x %x %x %n"); **CAN:** Overwrite important program flags that control access privileges, Overwrite return addresses on the stack, function pointers, etc //**Exploit**=program or technique that takes advantage of security vuln to violate explicit or implicit security policy // **Vulnerability=**set of conditions that allows attacker to violate an explicit or implicit security policy >> **Bugs=i**ntroduced during software implementation(buffer overflow) >>**Flaws=i**ntroduced during software design level(error handling problem)// **Mitigation**=methods/techniques/processes/tools that can prevent or limit exploits against vulnerabilities **@policy and procedure level**=background checks,type-safe lang **@system/network level**=multi-factor auth, network intrusion detection system **@source code level**=replacing unsafe functions // **Improve code by:** Know common security bugs/attack strategies, secure coding practice, source-code walkthroughs **Ex:**enforce all authenitication controls on server, pw hashing must be done on trusted system, use only HTTP POST requests to transmit auth credentials, all auth controls should fail securely // **SEVEN PERNICIOUS KINGDOMS:** help developers understand coding errors and recognize categories of problems that lead to vulnerabilities >> **1.Input validation and representation:** failure to account for untrusted input(ie:buff/int overflow, XSS,format string, SQL inj)>> TRUST BOUNDARY SECURITY:Programs must take steps to ensure data received across trust boundary can be appropriate, not malicious. Validation=make sure input data falls in boundary. Sanitation=ensure data conforms to reqs of subsystem to which its passed. Canonicalization=lossless reduction of input to equivalent simplest form. Normalization=conversion of input data to standard form.**Canon and norm before validating** **2.API abuse:** API establishes set of rules that callers and callees need to follow. Caller must check value returned from callee. **3.Security features:** Insecure use of security features can leave software system unprotected. Focus on authentication/access control/confidentiality/cryptography/privilege management **4.Time and state:** Unexpected interactions, failure to begin new session upon authentication. Be carfeul using functions that use filenames instead of file handles. **5.Errors:** Unhandled errors may have unexpected results and cause program to crash or terminate. Error handling should fix problem or notify user and terminate gracefully**don't expose sensitive info in exception messages** **6.Code Quality:** Poor quality leads to unpredictable behavior(ie:use of obsolete functions) **7.Encapsulation:** strong trust boundaries, make sure functionality or data doesnt cross trust boundaries and cause contamination(ie:escalation of data access privileges) ***Environment:** everything thats outside of source code but is still critical to security of product being created.// **Static Analysis:** examine text without attempting to execute, performed as part of code review. Most important is knowlege build into code of secure coding rules >> **Benefits**=reduce costs over system lifetime,educate devs about secure programming, automates repetitive and tedious aspects of source code checks >> **Issues**=rely on already known/fixed pattern of rules, human eval still required, doesnt detect design or architctural level flaws-only meant for source code, false sense of security // **Verification**=does software meet specification? **Validation**=does software meet user requirements? **Functional Testing Steps:** **1**.Identify requirements that softwares expected to perform 2.Create input test data based on functions specifications 3.Determine expected output test results based on functions specs 4.Execute test cases corresponding to functional reqs 5.Compare actual and expected outputs to determine functional compliance // **Black-box testing/functional testing**=tester only knows what software is supposed to do but cant see how it operates >> **Approaches to generate test inputs**= equivalence partitioning(reduce infinite set of test cases to few effective cases); boundary value analysis(test the boundaries) **White-box testing**=tester has access to program code and can examine it for clues to help with testing>> **Approaches to generate test inputs**= control flow testing(use program layout to make test cases, **code coverage analysis**=find areas of program not excersized by set of test cases, increase coverage); data flow testing=track piece of data throughout program // **Security Testing**=cover attackers mindset and security functionality // **White-hat testing**=test for pos, ensure features work as advertised>> **Test-to-pass=**ensure everything works>> **Functional Security Testing**=testing security mechanisms to ensure functionality is properly implemented, uses pos requirements(authentication mechanisms,error handling) // **Black-hat testing**=test for neg, ensure attacks cannot easily compromise system>> **Test-to-fail=**try to break program >> **Risk-based Security Testing**=simulates attackers approach(system shall not allow for data to be altered/destroyed or for system to be compromised/misused) >> **Penetration testing**=once software is complete and installed in op environment >> Steps: goal->info gathering->attack->report // **Unit Testing**=testing in coding phase /// **Malware=**instructions that run on victims comp that allow attacker to make system do what they want **ATTACK: VIRUS**=self replicating code that attaches to other programs(host) requiring you to click on it **COUNTER METHOD ANTIVIRUS** = works by static analysis(virus signature), heuristics(unseen viruses exhibiting certain behaviors), integrity verification(detecting unauthorized changes to file system) **Malware self preservation techniques:**stealthing(concealing presence of malware-preserve OG file size and pretend to be normal code), **obfuscation techniques:** dead-code insertion(inserting nop to change appearance), register reassignment(switching regs from generations), subroutine-reordering, instruction substitution(replacing some instr w equivalent ones), code transposition(reorder sequence of inst of original code), code integration(integrate code to code of target program) **ATTACK: WORM**=self-replicating code spreads via networks and you dont have to click on it, faster to take over vast number of machines, harder to trace back, increase damage **Elements of a worm:**1.gain access to target system through exploits(buffer overflow,file-sharing,email) 2.transfer to target 3. look for new victims(thorough email,random IPs, host lists, network neighbors) 4. scan victims for vulns 5. execute payload(consume bandwidth,open backdoor,DOS,crack encryption keys,remove files/deface website) **ATTACK: BACKDOOR**=program that allows attackers to bypass normal security controls on system gaining access on the attackers terms (local escalation of privileges, remote execution of commands, gain command shell, control GUI of victim remotely) run with permission of user **How to install backdoors:** exploits(buff overflows,system misconfigs),auto programs(viruses,worms),trick into installation (email,file-sharing) **ATTACK: TROJAN HORSES**=program that appears to have useful or benign purpose but really masks hidden malicious functionality, tricks user to download and install **To Trojanize software:** change name,combine backdoor with legit program using wrapper, insert code into software product during software development and testing **ATTACK: ROOTKITS:** trojan horse backdoor tools that modify existing OS software so an attacker can keep access to and hide on machine **DEFENSE AGAINST MALWARE=**deploy patches,update AV signatures,block arbitrary outbound connections; for devs=awareness of problems, build quality assurance into software dev