|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
|  |  |  | |  |
| Buffer overflow report  adam logan | | |
| 10/02/2023 | |  |

|  |
| --- |
|  |
| Table of contents [Analysis of Malware 3](#_Toc126936694)  [1.1 Executive Summary 3](#_Toc126936695)  [1.2 Testing Performed 3](#_Toc126936696)  [1.3 Vulnerabilities Detected 3](#_Toc126936697)  [1.4 Mitigation Techniques 4](#_Toc126936698)  [1.5 Actions Taken 4](#_Toc126936699)  [1.6 Preventative Measures 4](#_Toc126936700)  [1.7 Conclusion 4](#_Toc126936701)  [1.8 Appendices 5](#_Toc126936702)  [1.8.1 Misuse Case 5](#_Toc126936703)  [1.8.1.1 Misuse Case Description 5](#_Toc126936704)  [1.8.1.2 Misuse Case Diagram 5](#_Toc126936705)  [1.8.2 Normal Execution 6](#_Toc126936706)  [1.8.3 Checking Vulnerabilities 6](#_Toc126936707)  [1.8.4 Location Function and EIP 7](#_Toc126936708)  [1.8.5 The Exploit 8](#_Toc126936709)  [1.8.5.1 Initial Overflow 8](#_Toc126936710)  [1.8.5.2 Attempt to Overflow ‘Hacked’ 9](#_Toc126936711)  [1.8.6 Disassembly 10](#_Toc126936712)  [1.8.6.1 Disassembly Using ‘gdb-pwndbg’ 10](#_Toc126936713)  [1.8.6.1.1 ‘main’ method disassembly 10](#_Toc126936714)  [1.8.6.1.2 ‘Hacked’ method disassembly 11](#_Toc126936715)  [1.8.6.1.3 ‘displayStack’ method disassembly 13](#_Toc126936717)  [1.8.6.2 Disassembly Using ‘ghidra’ 14](#_Toc126936718)  [1.8.6.2.1 ‘main’ method disassembly 14](#_Toc126936719)  [1.8.6.2.2 ‘Hacked’ method disassembly 15](#_Toc126936720)  [1.8.6.2.3 ‘displayStack’ method disassembly 16](#_Toc126936721)  [1.8.7 Code Rewrite 17](#_Toc126936722)  [1.9 Bibliography 18](#_Toc126936723) | |

|  |  |
| --- | --- |
|  |  |
| Analysis of Malware1.1 Executive Summary This paper demonstrates that the product is not safe to put into production and will outline the methodology used, the vulnerability and what steps need to be taken to both mitigate the issue and prevent this.  A methodical approach was taken to testing the product ‘bufferoverflow’. Common tools such as checksec, gdb-pwndbg and ghidra were used. The product was disassembled, and multiple attempts were made to overflow the buffer.  The vulnerability at hand is a buffer overflow which can be used to control the flow of the program. This is a huge risk to the business but does not have a great cost to resolve.  What is needed is for security to be embedded at all stages of the software development lifecycle. A way to do this is by applying security within the artefacts [5]. As detailed within this paper, a code rewrite will need to be done as an unsecure function is used. Mitigation techniques will also need to be implemented, such as stack canaries and ASLR.  A misuse-case diagram can be found in section 1.8.1. 1.2 Testing Performed The first step was simply to see the normal execution of the program by passing a simple argument to the product and examining the result. As can be seen in figure 1.8.2.1 the program states that it is susceptible to a buffer overflow attack and that we must call a function at the address 0x080491A2.  The command checksec can be used to view the security measures that have been implemented. As we can see in figure 1.8.3.1 stack canaries, NX and ASLR have all been disabled and therefore the product may be susceptible to a buffer overflow.  At this point the tool gdb-pwndbg can be used in conjunction with cyclic to determine the length each register is overflown at (figure 1.8.4.3). The EIP register is overflown at 20 bytes (figure 1.8.4.4) and therefore the execution order can be modified.  The info function command within gdb-pwndbg can be used to find all the addresses of the functions (figure 1.8.4.2) and we can see the function Hacked at 0x080491A2 which is the same address stated earlier. At this stage we can use the command ./bufferoverflow $(python2 -c 'print("A"\*20+"\xa2\x91\x04\x08")')and we can see that Hacked is called (figure 1.8.5.1.1). The same process is applied to the input for Hacked but an overflow does not occur (section 1.8.5.2). | |
| 1.3 Vulnerabilities Detected As stated previously, there is a buffer overflow vulnerability. This severe vulnerability allows a malicious actor to execute functions which are not called, both the functions within the program and the functions within the DLLs.  Another risk is that shellcode could be placed within the buffer, with the instruction pointer, pointing back to the buffer, rather than to another function. This, used in conjunction with a NOP sled [8], could result in the user gaining root access, if the correct shellcode is used [7].  This is not a risk within this product as the shellcode size must be less than 20 bytes and the smallest, currently known, shellcode that can gain root access is 25 bytes [3], not accounting for a NOP sled, which would increase the likelihood of success. | |  |
|  |  |
| 1.4 Mitigation Techniques A possible mitigation step is stack canaries [6]. This technique is when a known value is put in between the data and the registers, and if this value is changed, then the compiler will halt the execution of the program [9]. Although these are not perfect as they can be bypassed [2] [4]  Enabling NX (no-execute bit)/DEP (Data Execution Prevention) would increase the reduction in threat level. As the stack should not contain any executable code [9], this area in memory could be marked as non-executable. Therefore, shellcode injected into the buffer would not be able to execute. As mentioned in section 1.3 most shellcodes cannot fit into the buffer and therefore another technique is needed.  A stronger technique than this is ASLR (Address Space Layout Randomization) which randomises the locations of instructional memory, which makes guessing these locations more difficult [6], and an attack that worked once may not work again [9]. Once again this is not impenetrable [4].  Using all these in conjunction will reduce the threat level, massively. 1.5 Actions Taken The disassembler gihdra and disassemble in gdb-pwndbg were used to gain a better understanding of the product (section 1.8.6). As can be seen within ghidra the product uses strcpy (figure 1.8.6.2.3.2). This is an unbounded memory function and therefore is the likely point of the vulnerability [10]. There are other functions like this, such as, sprintf, strcat and gets.  We can use bound checking [1] alternatives, such as, strncpy or strcpy\_s which both take, as an argument, the number of bytes to copy. Alternatives for the aforementioned functions are snprintf, strncat and fgets.  A possible action could be to remove Hacked as it is not used anywhere within this product. This would not be a viable solution if the product is imported by another program that uses Hacked.  Examples of rewritten can be found in 1.8.7. 1.6 Preventative Measures Security measures should have been implemented throughout the development lifecycle of this product as this would have prevented the vulnerability. A technique to do this is by requiring security elements within various artefacts [5].  An example security element that can be embedded within design and test artefacts is risk analysis. If this was carried out within the design stage, this may have prevented the vulnerability as C would have been identified as a memory unsafe language [4]. Possible alternatives could have been suggested at this point. 1.7 Conclusion This paper has demonstrated there is a buffer overflow vulnerability that can be exploited. An easy and simple way to detect this vulnerability is to check if the functions mentioned in 1.5 are used [6], by performing code reviews [5] during development.  A weakness in the report is the focus on the buffer overflow rather than investigating other vulnerabilities.  The suggestion of this paper is to incorporate security within the development lifecycle to prevent future vulnerabilities. Mitigation techniques must also be implemented to ensure that vulnerable assets are not exposed to malicious actors and, due to the functions used, a code rewrite should occur. | |
|  | |  |

|  |
| --- |
|  |
| 1.8 Appendices1.8.1 Misuse Case1.8.1.1 Misuse Case Description Within this section a misuse case diagram is shown, displaying both the use case of a regular user and that of a malicious actor. The malicious use cases are highlighted in black. As mentioned within the main body of the report a buffer overflow vulnerability can lead to many exploits, which are shown below. It is possible, and likely that malicious actor(s) would attempt all these exploits. 1.8.1.2 Misuse Case Diagram | |
| Figure 1.8.1.2.1 A misuse case diagram for the product | |

|  |  |
| --- | --- |
|  |  |
| 1.8.2 Normal Execution Graphical user interface, text  Description automatically generated   |  | | --- | | Figure 1.8.2.1 The normal execution of the program |  1.8.3 Checking Vulnerabilities   Figure 1.8.3.1 Using checksec to see what vulnerabilities  the product may have | |

|  |  |
| --- | --- |
|  |  |
| 1.8.4 Location Function and EIP   **Figure 1.8.4.1 Using** gdb-pwndbg **to get access to debugging tools**    **Figure 1.8.4.2 The result of**  info function**, which lists the**  **location of each function**    **Figure 1.8.4.3 The values within the registers when the command** run$(cyclic -100) **is used**    **Figure 1.8.4.4 The number of bytes until the EIP register is overflown** | |
|  | |  |

|  |
| --- |
|  |
| 1.8.5 The Exploit1.8.5.1 Initial Overflow   Figure 1.8.5.1.1 The execution of theHacked **function using the command**  ./bufferoverflow $(python2 -c 'print("A"\*20+"\xa2\x91\x04\x08")') | |

|  |
| --- |
|  |
| 1.8.5.2 Attempt to Overflow ‘Hacked’   **Figure 1.8.5.2.1** **Result of valid input (“Adam Logan 40293585”) into the** Hacked **function**    **Figure 1.8.5.2.2 Register values when valid input (“Adam Logan 40293585”) is used in the**  Hacked **function**    **Figure 1.8.5.2.3 Result of** cyclic500 **input into the** Hacked **function, in which the EIP**  **register has not been overflown** | |
|  |
| 1.8.6 Disassembly1.8.6.1 Disassembly Using ‘gdb-pwndbg’ **1.8.6.1.1 ‘main’ method disassembly** | |

**Figure 1.8.6.1.1.1 The** main **function disassembled using** gdb-pwnd

|  |
| --- |
|  |
| **1.8.6.1.2 ‘Hacked’ method disassembly** | |

|  |
| --- |
|  |
| A picture containing text  Description automatically generated **Figure 1.8.6.1.2.1 The** Hacked **function disassembled using** gdb-pwndbg | |

|  |
| --- |
|  |
| **1.8.6.1.3 ‘displayStack’ method disassembly** | |

**Figure 1.8.6.1.3.1 The** displayStack **function disassembled using** gdb-pwndbg

|  |
| --- |
|  |
| 1.8.6.2 Disassembly Using ‘ghidra’ **1.8.6.2.1 ‘main’ method disassembly**    **Figure 1.8.6.2.1.1 The** main **function disassembled using** ghidra    **Figure 1.8.6.2.1.2 Estimated C code for the** main **function using** ghidra | |

|  |
| --- |
|  |
| **1.8.6.2.2 ‘Hacked’ method disassembly**    **Figure 1.8.6.2.2.1 The** Hacked **function disassembled using** ghidra    **Figure 1.8.6.2.2.2 Estimated C code for the** Hacked **function using** ghidra | |

|  |
| --- |
|  |
| **1.8.6.2.3 ‘displayStack’ method disassembly**    **Figure 1.8.6.2.3.1 The** displayStack **function disassembled using** ghidra    **Figure** 1.8.6.2.3.2 **Estimated C code for the** displayStack **function using** ghidra | |
|  |
| 1.8.7 Code Rewrite —————————————————————————————————————————————————  void displayStack(const char input[]){  char buf[8];  printf("--------------------------------------\n");  printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t  \t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");  strncpy(buf, input, (int) sizeof buf); // or strcpy\_s(buf, (int) sizeof buf, input);  printf("\nBuffer \n\t\t\t%s\n\n", buf);  printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t  \t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");  printf("--------------------------------------\n");  }  —————————————————————————————————————————————————  **Listing 1.8.7.1 An example of the secure alternative to** strcpy**,** strncpy  —————————————————————————————————————————————————  int displayStack(const char input[]){      char buf[8];      printf("--------------------------------------\n");     printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t  \t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");      int i = 0;      while(input[i] != 0) {          if(input[i] >= 128 || input[i] < 0) {              printf("You have entered a non-ascii character and the program has exited");              return 0;          }          printf("char num %d \n", i);          i++;      }      strncpy(buf, input, (int) sizeof buf);      printf("\nBuffer \n\t\t\t%s\n\n", buf);     printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t  \t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");      printf("--------------------------------------\n");      return 1;  }  —————————————————————————————————————————————————  Listing **1.8.7.2** The same code from 1.8.7.1 but a while loop has been added to prevent shellcode from being  written into the buffer | |
|  |
| 1.9 Bibliography  1. Andress, Jason. “Buffer Overflows”. *Foundations of Information Security: A Straightforward Introduction*. No Starch Press, 2019, pp. 175. 2. CryptoCat. “Bypassing Stack Canaries.” *YouTube*, 10 August 2022, <https://www.youtube.com/watch?v=TOImpHQvmpo&list=PLHUKi1UlEgOIc07Rfk2Jgb5fZbxDPec94&index=12> Accessed 7 February 2023. 3. Erickson, Jon. “Shell-Spawning Shellcode.” *Hacking: The Art of Exploitation, 2nd Edition*, No Starch Press, 2008, p. 298. 4. Forshaw, James. *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*. No Starch Press, 2017. pp. 5. McGraw, Gary. “Software Security.” *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80 - 83. *Software security*, <https://ieeexplore.ieee.org/document/1281254> Accessed 2 February 2023. 6. Peguero, Ksenia, and Vineeta Sangaraju. “How to detect, prevent, and mitigate buffer overflow attacks.” *Synopsys*, 7 February 2017, <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/> Accessed 7 February 2023. 7. Pound, Mike. “Running a Buffer Overflow Attack.” *YouTube*, 10 August 2022, <https://www.youtube.com/watch?v=1S0aBV-Waeo&t=646s> Accessed 7 February 2023. 8. Sikorski, Michael, and Andrew Honig. “NOP Sled.” *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 2012, pp. 422 - 423. 9. Watters, Brendan. “Stack-Based Buffer Overflow Attacks: Explained.” *Rapid7*, 19 February 2019, <https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/>   Accessed 7 February 2023.   1. Yaworski, Peter. “Buffer Overflows.” *Real-World Bug Hunting: A Field Guide to Web Hacking*, No Starch Press, 2019, pp. 130 - 133. | |