Buffer overflow - Wikipedia, the free encyclopedia

In computer security and programming, a **buffer overflow**, or **buffer overrun**, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety.

Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited.

Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows.

Contents

[hide]

Technical description[edit][|]edit source]

A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer due to insufficient bounds checking. This can occur when copying data from one buffer to another without first checking that the data fits within the destination buffer.

Example[edit][|]edit source]

In the following example, a program has two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte big-endian integer, B.

```
char           A[8];
unsigned short B;
```

Initially, A contains nothing but zero bytes, and B contains the number 1979.

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

Now, the program attempts to store the null-terminated string "excessive" with ASCII encoding in the A buffer.

```
strcpy(A, "excessive");
```

"excessive" is 9 characters long and encodes to 10 bytes including the terminator, but A can take only 8 bytes. By failing to check the length of the string, it also overwrites the value of B:

variable name	A	B

value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

B's value has now been inadverently replaced by a number formed from part of the character string. In this example "e" followed by a zero byte would become 25856.

Writing data past the end of allocated memory can sometimes be detected by the operating system to generate a segmentation fault error that terminates the process.

For more details on stack-based overflows, see Stack buffer overflow.

Exploitation[edit]|]edit source]

The techniques to exploit a buffer overflow vulnerability vary per architecture, operating system and memory region. For example, exploitation on the heap (used for dynamically allocated memory), is very different from exploitation on the call stack.

Stack-based exploitation[edit]|]edit source]

Main article: Stack buffer overflow

A technically inclined user may exploit stack-based buffer overflows to manipulate the program to their advantage in one of several ways:

- By overwriting a local variable that is near the buffer in memory on the stack to change the behavior of the program which may benefit the attacker.
- By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer.
- By overwriting a function pointer,[1] or exception handler, which is subsequently executed.
- By overwriting a parameter of a different stack frame or a non local address pointed to in the current stack context.[2]

With a method called "trampolining", if the address of the user-supplied data is unknown, but the location is stored in a register, then the return address can be overwritten with the address of an opcode which will cause execution to jump to the user supplied data. If the location is stored in a register R, then a jump to the location containing the opcode for a jump R, call R or similar instruction, will cause execution of user supplied data. The locations of suitable opcodes, or bytes in memory, can be found in DLLs or the executable itself. However the address of the opcode typically cannot contain any null characters and the locations of these opcodes can vary between applications and versions of the operating system. The Metasploit Project is one such database of suitable opcodes, though only those found in the Windows operating system are listed.[3]

Stack-based buffer overflows are not to be confused with stack overflows.

Also note that these vulnerabilities are usually discovered through the use of a fuzzer.[4]

Heap-based exploitation[edit]|]edit source]

Main article: Heap overflow

A buffer overflow occurring in the heap data area is referred to as a heap overflow and is exploitable in a manner different from that of stack-based overflows. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc meta data) and uses the resulting pointer exchange to overwrite a program function pointer.

Microsoft's GDI+ vulnerability in handling JPEGs is an example of the danger a heap overflow can present.[5]

Barriers to exploitation[edit][]edit source]

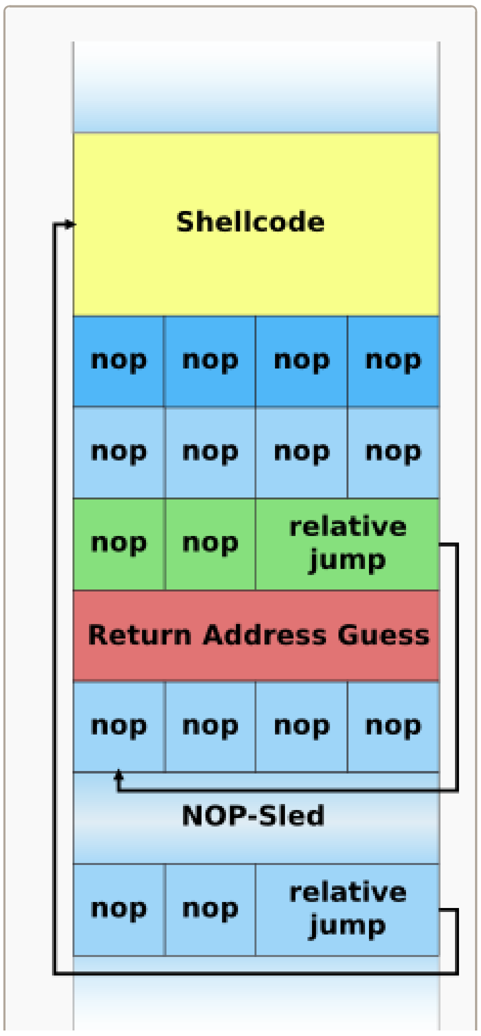
Manipulation of the buffer, which occurs before it is read or executed, may lead to the failure of an exploitation attempt. These manipulations can mitigate the threat of exploitation, but may not make it impossible. Manipulations could include conversion to upper or lower case, removal of metacharacters and filtering out of non-alphanumeric strings. However, techniques exist to bypass these filters and manipulations; alphanumeric code, polymorphic code, self-modifying code and return-to-libc attacks. The same methods can be used to avoid detection by intrusion detection systems. In some cases, including where code is converted into unicode,[6] the threat of the vulnerability have been misrepresented by the disclosers as only Denial of Service when in fact the remote execution of arbitrary code is possible.

Practicalities of exploitation[edit][]edit source]

In real-world exploits there are a variety of challenges which need to be overcome for exploits to operate reliably. These factors include null bytes in addresses, variability in the location of shellcode, differences between environments and various counter-measures in operation.

NOP sled technique[edit][]edit source]

Main article: NOP slide



A NOP-sled is the oldest and most widely known technique for successfully exploiting a stack buffer overflow.[7] It solves the problem of finding the exact address of the buffer by effectively increasing the size of the target area. To do this, much larger sections of the stack are corrupted with the no-op machine instruction. At the end of the attacker-supplied data, after the no-op instructions, the attacker places an instruction to perform a relative jump to the top of the buffer where the shellcode is located. This collection of no-ops is referred to as the "NOP-sled" because if the return address is overwritten with any address within the no-op region of the buffer it will "slide" down the no-ops until it is redirected to the actual malicious code by the jump at the end. This technique requires the attacker to guess where on the stack the NOP-sled is instead of the comparatively small shellcode.[8]

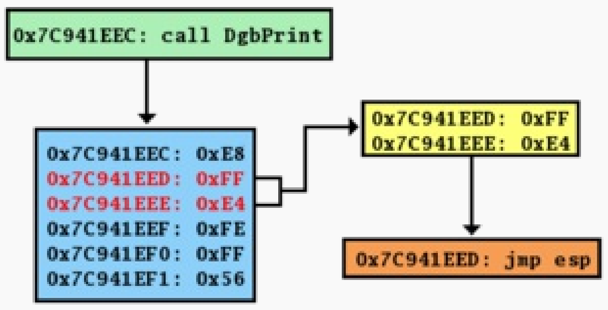
Because of the popularity of this technique, many vendors of intrusion prevention systems will search for this pattern of no-op machine instructions in an attempt to detect shellcode in use. It is important to note that a NOP-sled does not necessarily contain only traditional no-op machine instructions; any instruction that does not corrupt the machine state to a point where the shellcode will not run can be used in place of the hardware assisted no-op. As a result it has become common practice for exploit writers to compose the no-op sled with randomly chosen instructions which will have no real effect on the shellcode execution.[9]

Illustration of a NOP-sled payload on the stack.

While this method greatly improves the chances that an attack will be successful, it is not without problems. Exploits using this technique still must rely on some amount of luck that they will guess offsets on the stack that are within the NOP-sled region.[10] An incorrect guess will usually result in the target program crashing and could alert the system administrator to the attacker's activities. Another problem is that the NOP-sled requires a much larger amount of memory in which to hold a NOP-sled large enough to be of any use. This can be a problem when the allocated size of the affected buffer is too small and the current depth of the stack is shallow (i.e. there is not much space from the end of the current stack frame to the start of the stack). Despite its problems, the NOP-sled is often the only method that will work for a given platform, environment, or situation; as such it is still an important technique.

The jump to address stored in a register technique[edit]|]edit source]

The "jump to register" technique allows for reliable exploitation of stack buffer overflows without the need for extra room for a NOP-sled and without having to guess stack offsets. The strategy is to overwrite the return pointer with something that will cause the program to jump to a known pointer stored within a register which points to the controlled buffer and thus the shellcode. For example, if register A contains a pointer to the start of a buffer then any jump or call taking that register as an operand can be used to gain control of the flow of execution.[11]



An instruction from ntdll.dll to call the DbgPrint() routine contains the i386 machine opcode for jmp esp.

In practice a program may not intentionally contain instructions to jump to a particular register. The traditional solution is to find an unintentional instance of a suitable opcode at a fixed location somewhere within the program memory. In figure E on the left you can see an example of such an unintentional instance of the i386 jmp esp instruction. The opcode for this instruction is FF E4 .[12] This two byte sequence can be found at a one byte offset from the start of the instruction call DbgPrint at address 0x7C941EED. If an attacker overwrites the program return address with this address the program will first jump to 0x7C941EED, interpret the opcode FF E4 as the jmp esp instruction, and will then jump to the top of the stack and execute the attacker's code.[13]

When this technique is possible the severity of the vulnerability increases considerably. This is because exploitation will work reliably enough to automate an attack with a virtual guarantee of success when it is run. For this reason, this is the technique most commonly used in Internet worms that exploit stack buffer overflow vulnerabilities.[14]

This method also allows shellcode to be placed after the overwritten return address on the Windows platform. Since executables are mostly based at address 0x00400000 and x86 is a Little Endian architecture, the last byte of the return address must be a null, which terminates the buffer copy and nothing is written beyond that. This limits the size of the shellcode to the size of the buffer, which may be overly restrictive. DLLs are located in high memory (above 0x01000000) and so have addresses containing no null bytes, so this method can remove null bytes (or other disallowed characters) from the overwritten return address. Used in this way, the method is often referred to as "DLL Trampolining".

Protective countermeasures[edit]|]edit source]

Various techniques have been used to detect or prevent buffer overflows, with various tradeoffs. The most reliable way to avoid or prevent buffer overflows is to use automatic protection at the language level.

This sort of protection, however, cannot be applied to legacy code, and often technical, business, or cultural constraints call for a vulnerable language. The following sections describe the choices and implementations available.

Choice of programming language[edit][|]edit source]

The choice of programming language can have a profound effect on the occurrence of buffer overflows. As of 2008[update], among the most popular languages are C and its derivative, C++, with a vast body of software having been written in these languages. C and C++ provide no built-in protection against accessing or overwriting data in any part of memory; more specifically, they do not check that data written to a buffer is within the boundaries of that buffer. However, the standard C++ libraries provide many ways of safely buffering data, and techniques to avoid buffer overflows also exist for C.

Many other programming languages provide runtime checking and in some cases even compile-time checking which might send a warning or raise an exception when C or C++ would overwrite data and continue to execute further instructions until erroneous results are obtained which might or might not cause the program to crash. Examples of such languages include Ada, Eiffel, Lisp, Modula-2, Smalltalk, OCaml and such C-derivatives as Cyclone, Rust and D. The Java and .NET Framework bytecode environments also require bounds checking on all arrays. Nearly every interpreted language will protect against buffer overflows, signalling a well-defined error condition. Often where a language provides enough type information to do bounds checking an option is provided to enable or disable it. Static code analysis can remove many dynamic bound and type checks, but poor implementations and awkward cases can significantly decrease performance. Software engineers must carefully consider the tradeoffs of safety versus performance costs when deciding which language and compiler setting to use.

Use of safe libraries[edit][|]edit source]

The problem of buffer overflows is common in the C and C++ languages because they expose low level representational details of buffers as containers for data types. Buffer overflows must thus be avoided by maintaining a high degree of correctness in code which performs buffer management. It has also long been recommended to avoid standard library functions which are not bounds checked, such as `gets`, `scanf` and `strcpy`. The Morris worm exploited a `gets` call in fingerd.[15]

Well-written and tested abstract data type libraries which centralize and automatically perform buffer management, including bounds checking, can reduce the occurrence and impact of buffer overflows. The two main building-block data types in these languages in which buffer overflows commonly occur are strings and arrays; thus, libraries preventing buffer overflows in these data types can provide the vast majority of the necessary coverage. Still, failure to use these safe libraries correctly can result in buffer overflows and other vulnerabilities; and naturally, any bug in the library itself is a potential vulnerability. "Safe" library implementations include "The Better String Library",[16] Vstr [17] and Erwin.[18] The OpenBSD operating system's C library provides the strlcpy and strlcat functions, but these are more limited than full safe library implementations.

In September 2007, Technical Report 24731, prepared by the C standards committee, was published;[*citation needed*] it specifies a set of functions which are based on the standard C library's string and I/O functions, with additional buffer-size parameters. However, the efficacy of these functions for the purpose of reducing buffer overflows is disputable; it requires programmer intervention on a per function call basis that is equivalent to intervention that could make the analogous older standard library functions buffer overflow safe.[19]

Buffer overflow protection[edit][|]edit source]

Main article: Buffer overflow protection

Buffer overflow protection is used to detect the most common buffer overflows by checking that the stack has not been altered when a function returns. If it has been altered, the program exits with a segmentation fault. Three such systems are Libsafe,[20] and the *StackGuard*[21] and *ProPolice*[22] gcc patches.

Microsoft's Data Execution Prevention mode explicitly protects the pointer to the SEH Exception Handler from being overwritten.[23]

Stronger stack protection is possible by splitting the stack in two: one for data and one for function returns. This split is present in the Forth language, though it was not a security-based design decision. Regardless, this is not a complete solution to buffer overflows, as sensitive data other than the return address may still be overwritten.

Pointer protection[edit][|]edit source]

Buffer overflows work by manipulating pointers (including stored addresses). PointGuard was proposed as a compiler-extension to prevent attackers from being able to reliably manipulate pointers and addresses.[24] The approach works by having the compiler add code to automatically XOR-encode pointers before and after they are used. Because the attacker (theoretically) does not know what value will be used to encode/decode the pointer, he cannot predict what it will point to if he overwrites it with a new value. PointGuard was never released, but Microsoft implemented a similar approach beginning in Windows XP SP2 and Windows Server 2003 SP1.[25] Rather than implement pointer protection as an automatic feature, Microsoft added an API routine that can be called at the discretion of the programmer. This allows for better performance (because it is not used all of the time), but places the burden on the programmer to know when it is necessary.

Because XOR is linear, an attacker may be able to manipulate an encoded pointer by overwriting only the lower bytes of an address. This can allow an attack to succeed if the attacker is able to attempt the exploit multiple times and/or is able to complete an attack by causing a pointer to point to one of several locations (such as any location within a NOP sled).[26] Microsoft added a random rotation to their encoding scheme to address this weakness to partial overwrites.[27]

Executable space protection[edit][|]edit source]

Main article: Executable space protection

Executable space protection is an approach to buffer overflow protection which prevents execution of code on the stack or the heap. An attacker may use buffer overflows to insert arbitrary code into the memory of a program, but with executable space protection, any attempt to execute that code will cause an exception.

Some CPUs support a feature called NX ("No eXecute") or XD ("eXecute Disabled") bit, which in conjunction with software, can be used to mark pages of data (such as those containing the stack and the heap) as readable and writeable but not executable.

Some Unix operating systems (e.g. OpenBSD, Mac OS X) ship with executable space protection (e.g. W^X). Some optional packages include:

Newer variants of Microsoft Windows also support executable space protection, called Data Execution Prevention.[31] Proprietary add-ons include:

- BufferShield[32]
- StackDefender[33]

Executable space protection does not generally protect against return-to-libc attacks, or any other attack which does not rely on the execution of the attackers code. However, on 64-bit systems using ASLR, as described below, executable space protection makes it far more difficult to execute such attacks.

Address space layout randomization[edit]|]edit source]

Main article: Address space layout randomization

Address space layout randomization (ASLR) is a computer security feature which involves arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, randomly in a process' address space.

Randomization of the virtual memory addresses at which functions and variables can be found can make exploitation of a buffer overflow more difficult, but not impossible. It also forces the attacker to tailor the exploitation attempt to the individual system, which foils the attempts of internet worms.[34] A similar but less effective method is to rebase processes and libraries in the virtual address space.

Deep packet inspection[edit]|]edit source]

Main article: Deep packet inspection

The use of deep packet inspection (DPI) can detect, at the network perimeter, very basic remote attempts to exploit buffer overflows by use of attack signatures and heuristics. These are able to block packets which have the signature of a known attack, or if a long series of No-Operation instructions (known as a nop-sled) is detected, these were once used when the location of the exploit's payload is slightly variable.

Packet scanning is not an effective method since it can only prevent known attacks and there are many ways that a 'nop-sled' can be encoded. Shellcode used by attackers can be made alphanumeric, metamorphic, or self-modifying to evade detection by heuristic packet scanners and intrusion detection systems.

History[edit]|]edit source]

Buffer overflows were understood and partially publicly documented as early as 1972, when the Computer Security Technology Planning Study laid out the technique: "The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine." (Page 61)[35] Today, the monitor would be referred to as the kernel.

The earliest documented hostile exploitation of a buffer overflow was in 1988. It was one of several exploits used by the Morris worm to propagate itself over the Internet. The program exploited was a service on Unix called finger.[36] Later, in 1995, Thomas Lopatic independently rediscovered the buffer overflow and published his findings on the Bugtraq security mailing list.[37] A year later, in 1996, Elias Levy (aka Aleph One) published in *Phrack* magazine the paper "Smashing the Stack for Fun and Profit",[38] a step-by-step introduction to exploiting stack-based buffer overflow vulnerabilities.

Since then, at least two major internet worms have exploited buffer overflows to compromise a large number of systems. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services (IIS) 5.0[39] and in 2003 the SQL Slammer worm compromised machines running Microsoft SQL Server 2000.[40]

In 2003, buffer overflows present in licensed Xbox games have been exploited to allow unlicensed software, including homebrew games, to run on the console without the need for hardware modifications, known as modchips.[41] The PS2 Independence Exploit also used a buffer overflow to achieve the same for the PlayStation 2. The Twilight hack accomplished the same with the Wii, using a buffer overflow in *The Legend of Zelda: Twilight Princess*.

See also[edit]|]edit source]

Notes[edit]|]edit source]

1. ^ "CORE-2007-0219: OpenBSD's IPv6 mbufs remote kernel buffer overflow". Retrieved 2007-05-15.
2. ^ "Modern Overflow Targets". Retrieved 2013-07-05.
3. ^ "The Metasploit Opcode Database". Retrieved 2007-05-15. [*dead link*]
4. ^ "The Exploitant - Security info and tutorials". Retrieved 2009-11-29.
5. ^ "Microsoft Technet Security Bulletin MS04-028". Retrieved 2007-05-15.
6. ^ "Creating Arbitrary Shellcode In Unicode Expanded Strings" (PDF). Retrieved 2007-05-15.
7. ^ Vangelis (2004-12-08). *Stack-based Overflow Exploit: Introduction to Classical and Advanced Overflow Technique* (text). Wowhacker via Neworder. [*dead link*]
8. ^ Balaban, Murat. *Buffer Overflows Demystified* (text). Enderunix.org. [*dead link*]
9. ^ Akritidis, P.; Evangelos P. Markatos, M. Polychronakis, and Kostas D. Anagnostakis (2005). "STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis." (PDF). *Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC 2005)*. IFIP International Information Security Conference. Retrieved 2012-03-04.
10. ^ Klein, Christian (2004-09). *Buffer Overflow* (PDF).
11. ^ Shah, Saumil (2006). "Writing Metasploit Plugins: from vulnerability to exploit" (PDF). *Hack In The Box*. Kuala Lumpur. Retrieved 2012-03-04.
12. ^ *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M* (PDF). Intel Corporation. 2007-05. pp. 3–508. [*dead link*]
13. ^ Alvarez, Sergio (2004-09-05). *Win32 Stack BufferOverFlow Real Life Vuln-Dev Process* (PDF). IT Security Consulting. Retrieved 2012-03-04.
14. ^ Ukai, Yuji; Soeder, Derek; Permeh, Ryan (2004). "Environment Dependencies in Windows Exploitation". *BlackHat Japan*. Japan: eEye Digital Security. Retrieved 2012-03-04.
15. ^ http://wiretap.area.com/Gopher/Library/Techdoc/Virus/inetvir.823
16. ^ "The Better String Library".
17. ^ "The Vstr Homepage". Retrieved 2007-05-15.
18. ^ "The Erwin Homepage". Retrieved 2007-05-15.
19. ^ "CERT Secure Coding Initiative". Retrieved 2007-07-30.
20. ^ "Libsafe at FSF.org". Retrieved 2007-05-20.
21. ^ "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks by Cowan et al." (PDF). Retrieved 2007-05-20.
22. ^ "ProPolice at X.ORG". Retrieved 2007-05-20.
23. ^ "Bypassing Windows Hardware-enforced Data Execution Prevention". Retrieved 2007-05-20.
24. ^ PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities
25. ^ Protecting Against Pointer Subterfuge (Kinda!)
26. ^ Defeating Compiler-Level Buffer Overflow Protection
27. ^ Protecting against Pointer Subterfuge (Redux)
28. ^ "PaX: Homepage of the PaX team". Retrieved 2007-06-03.
29. ^ "KernelTrap.Org". Retrieved 2007-06-03.
30. ^ "Openwall Linux kernel patch 2.4.34-ow1". Retrieved 2007-06-03.
31. ^ "Microsoft Technet: Data Execution Prevention".

32. ^ "BufferShield: Prevention of Buffer Overflow Exploitation for Windows". Retrieved 2007-06-03.
33. ^ "NGSec Stack Defender". Archived from the original on 2007-05-13. Retrieved 2007-06-03.
34. ^ "PaX at GRSecurity.net". Retrieved 2007-06-03.
35. ^ "Computer Security Technology Planning Study" (PDF). Retrieved 2007-11-02.
36. ^ ""A Tour of The Worm" by Donn Seeley, University of Utah". Archived from the original on 2007-05-20. Retrieved 2007-06-03.
37. ^ "Bugtraq security mailing list archive". Archived from the original on 2007-09-01. Retrieved 2007-06-03.
38. ^ ""Smashing the Stack for Fun and Profit" by Aleph One". Retrieved 2012-09-05.
39. ^ "eEye Digital Security". Retrieved 2007-06-03.
40. ^ "Microsoft Technet Security Bulletin MS02-039". Retrieved 2007-06-03.
41. ^ "Hacker breaks Xbox protection without mod-chip". Retrieved 2007-06-03.

External links[edit]|]edit source]

[hide]
Memory management

Manual memory management	
Virtual memory	
Hardware	• Memory management unit • Translation lookaside buffer
Garbage collection	
Memory segmentation	
Memory safety	• **Buffer overflow** • Dangling pointer • Stack overflow
Issues	
Other	