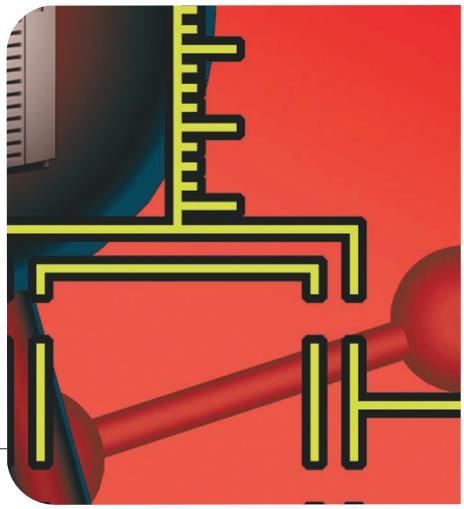
Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns



This article describes three powerful general-purpose families of exploits for buffer overruns: arc injection, pointer subterfuge, and heap smashing. These new techniques go beyond the traditional "stack smashing" attack and invalidate traditional assumptions about buffer overruns.

Jonathan Pincus
Microsoft Research

Brandon Baker
Microsoft

Security vulnerabilities related to buffer overruns account for the largest share of CERT advisories, as well as high-profile worms—from the original Internet Worm in 1987 through Blaster's appearance in 2003. When malicious crackers discover a vulnerability, they devise *exploits* that take advantage of the vulnerability to attack a system.

The traditional approach to exploiting buffer overruns is *stack smashing*: modifying a return address saved on the stack (the region of memory used for parameters, local variables, and return address) to point to code the attacker supplies that resides in a stack buffer at a known location. Discussions of buffer overrun exploitation in software engineering literature typically concentrate on stack-smashing attacks. As a result, many software engineers and even security professionals seemingly assume that all buffer overrun exploits operate in a similar manner.

During the past decade, however, hackers have developed several additional approaches to exploit buffer overruns. The *arc injection* technique (sometimes referred to as *return-into-libc*) involves a control transfer to code that already exists in the program's memory space. Various *pointer subterfuge* techniques change the program's control flow by attacking function pointers (pointer variables whose value is used as an address in a function call) as an alternative to the saved return address, or modify arbitrary memory locations to create more complex exploits by subverting data pointers. *Heap smashing* allows exploitation of buffer overruns in dynamically allocated memory, as opposed to on the stack.

While these techniques appear esoteric, they are in fact practical for real-world vulnerabilities. The Apache/Open_SSL Slapper worm[1] was the first high-profile worm to employ heap smashing.

The wide variety of exploits published for the stack buffer overrun that Microsoft Security Bulletin MS03-026[2] addressed (the vulnerability exploited by Blaster) further illustrates the viability of these new techniques. Table 1 shows a few examples.

These new approaches typically first surface in nontraditional hacker publications. (For more discussion of this, including references to the origins of many of these techniques, see the "Nontraditional literature" sidebar on page 26.) A few brief summaries in the mainstream literature,[3,4] and recent books (such as *Exploiting Software: How to Break Code*[5] and *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*[6]), also discuss some of these techniques.

Understanding new approaches to exploitation is vital for evaluating countermeasures' effectiveness in addressing the buffer overrun problem. Failure to consider them leads people to suggest overly simplistic and worthless "solutions" (for example, "move all the buffers to the heap") or to overstate the value of useful improvements (for instance, valuable mitigation methods such as a nonexecutable stack, which many mistakenly view as a silver bullet).

Exploiting buffer overruns

A buffer overrun occurs when a program attempts to read or write beyond the end of a bounded array (also known as a *buffer*). In runtime environments for languages such as Pascal, Ada, Java, and C#, the runtime environment de-

Table 1. Different exploits of Microsoft Security Bulletin MS03-026.

AUTHOR	TECHNIQUE	COMMENTS
Last Stage of Delirium	Unknown	Original report; specific exploit was never published
XFocus	Stack smashing	Apparently used by Blaster author
Litchfield	Pointer subterfuge	
Cigital	Pointer subterfuge	Unpublished
K-otic	Arc injection	
Authors of this article	Pointer subterfuge and arc injection	Unpublished

tects buffer overruns and generates an exception. In the runtime environment for C and C++, however, no such checking is performed. Attackers can often exploit the defect by using the buffer overrun to change the program's execution. Such exploits can in turn provide the infection vector for a worm or a targeted attack on a given machine.

A buffer overrun is characterized as a *stack buffer overrun* or *heap buffer overrun* depending on what memory gets overrun. C and C++ compilers typically use the stack for local variables as well as parameters, frame pointers, and saved return addresses. Heaps, in this context, refer to any dynamic memory implementations such as the C standard library's `malloc/free`, C++'s `new/delete`, or the Microsoft Windows APIs `HeapAlloc/HeapFree`. Figure 1 provides examples of functions containing a stack buffer overrun (a) and heap buffer overrun (b).

```
(a)
void f1a(void * arg, size_t len) {
  char buff[100];
  memcpy(buff, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}

(b)
void f1b(void * arg, size_t len) {
  char * ptr = malloc(100);
  if (ptr == NULL) return;
  memcpy(ptr, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}
```

Figure 1. Code samples with traditional (simple) buffer overrun defects. (a) A stack buffer overrun and (b) a heap buffer overrun.

Published general-purpose exploits for buffer overruns typically involve two steps:

1. Change the program's flow of control. (Pure data exploits in which the buffer happens to be adjacent to a security-critical variable operate without changing the program's flow of control; these are relatively rare, and to date no general-purpose techniques have been published relating to them.)
2. Execute some code (potentially supplied by the attacker) that operates on some data (also potentially supplied by the attacker).

The term *payload* refers to the combination of code or data that the attacker supplies to achieve a particular goal (for example, propagating a worm). The attacker sometimes provides the payload as part of the operation that causes the buffer overrun, but this need not be the case. All that is required is that the payload be at a known or discoverable location in memory at the time that the unexpected control-flow transfer occurs.

Stack smashing

Stack smashing, a technique first described in detail in the mid '90s by such hackers as AlephOne and DilDog, illustrates these two steps: changing the flow of control to execute attacker-supplied code. Stack smashing relies on the fact that most C compilers store the saved return address on the same stack used for local variables.

To exploit the buffer overrun in `f1a` via a classic stack-smashing attack, the attacker must supply a value for `arg` (for example, as received from a network packet) that contains both an executable payload and the address at which the payload will be loaded into the program's memory—that is, the address of `buff`. This value must be positioned at a location within `arg` that corresponds to where `f1a`'s return address is stored on the program's

stack when the buffer overrun occurs; depending on the specific compiler used to compile `f1a`, this will typically be somewhere around 108 bytes into `arg`. This new value for the saved return address changes the program's control flow: when the return instruction executes, control is transferred to `buff` instead of returning to the calling procedure.

Stack smashing is well described in mainstream literature,[1] so we won't discuss it in detail in this article. However, two important enhancements to the standard stack-smashing technique are often used in conjunction with some of the other approaches, and therefore are worth describing.

Trampolining lets an attacker apply stack smashing in situations in which `buff`'s absolute address is not known ahead of time. The key insight here is that if a program register *R* contains a value relative to `buff`, we can transfer control to `buff` by first transferring control to a sequence of instructions that indirectly transfers via *R*. When an attacker can find such a sequence of instructions (known as the trampoline) at a well-known or predictable address, this provides a reliable mechanism for transferring control to `buff`. Alternatively, a pointer to attacker-supplied data can often be found somewhere on the stack, and so exploits often use sequences of instructions such as `pop/pop/ret` to trampoline without involving registers. David Litchfield's comparison of exploit methods between Linux and Windows suggests that variations in Linux distributions make trampolines a less popular exploit approach on Linux.[6]

Another concept first developed as a stack-smashing enhancement is the separation between transferring the payload and the buffer overrun operation that modifies control flow. A situation in which this is particularly useful is when the buffer being overrun is too small to contain an attacker's payload. If the attacker arranges for the payload to be supplied in an earlier operation and it's still available in the program's memory space, then he or she can use it at the time of exploit. The sidebar briefly discusses and gives references for several such clever techniques, including Murat Balaban's approach of storing the payload in an environment variable, which are typically accessible on Linux systems from well-known addresses near the stack base.

Arc injection

As an alternative to supplying executable code, an attacker might simply be able to supply data that—when a program's existing code operates on it—will lead to the desired effect. One such example occurs if the attacker can supply a command line that the program under attack will use to spawn another process; this essentially allows arbitrary code execution. Arc injection exploits are an example of this data-oriented approach—indeed, the first such published exploit allowed the attacker to run an arbitrary program. The term "arc injection" refers to how these exploits operate: the exploit just inserts a new arc (control-flow transfer) into the program's control-flow graph, as opposed to code injection-exploits such as stack smashing, which also insert a new node.

A straightforward version of arc injection is to use a stack buffer overrun to modify the saved return address to point to a location already in the program's address space—more specifically, to a location within the `system` function in the C standard library. The `system` function takes an arbitrary command line as an argument, checks the argument's validity, loads it into a register *R*, and makes a system call to create the process. Eliding some details, pseudocode for the `system` function is:

```
void system(char * arg) {
  check_validity(arg);
  R = arg;
target:
  execl(R, ...)
}
```

If an attacker can arrange for *R* to point to an attacker-supplied string and then jump directly to the location `target`, thus bypassing the validity check and assignment, the system will treat the attacker-supplied string as a command line and execute it. On many operating systems, the C standard library loads into most processes at a well-known location, and the computing `target`'s absolute address is straightforward.

This still leaves the attacker with the problem of how to get *R* to point to an attacker-supplied string. In many cases, this is trivial: programs routinely reuse registers, and it could just so happen that the program also uses *R* in the procedure in which a buffer overrun occurs. For example, if *R* happens to contain the address of `buff` or `arg` in the compiled version `f1` (from Figure 1), then exploiting the buffer overflow via arc injection is trivial: all the attacker needs to do is ensure that the `target`'s location appears at the appropriate offset in `arg` to replace the saved return address. When `f1` returns, control transfers to the middle of the `system` function instead. (Because `system` is in the C standard library, known as `libc` on Unix and Linux systems, these exploits are often referred to as return-into-libc exploits; however, variations exist that involve neither a return statement nor libc, so we prefer the more general term "arc injection.")

The straightforward arc-injection approach serves as the basis for more complex exploits. One approach is to arrange the data after the saved return address so that `f1` first "returns" to `strcpy`'s location rather than its original caller (copying the data to an appropriate location);

`strcpy` then "returns" to `system`. Generalizing this allows for arbitrary "chaining" of function calls in arc-injection exploits.

Arc-injection exploits are especially useful techniques when the program being attacked has some form of memory protection (for example, nonexecutable stacks or the so-called W^X mechanism, which prevents any area of memory from being simultaneously writeable and executable). Because no attacker-supplied code is executed, these mitigations do not prevent arc-injection exploits.

Pointer subterfuge

Pointer subterfuge is a general term for exploits that involve modifying a pointer's value. At least four varieties of pointer subterfuge exist: function-pointer clobbering, data-pointer modification, exception-handler hijacking, and virtual pointer (VPTR) smashing. Actual implementation of these depends to some extent on how the compiler lays out local variables and parameters; for the sake of simplicity, we will ignore such complications in this article.

Function-pointer clobbering

Function-pointer clobbering is exactly what it sounds like: modifying a function pointer to point to attacker-supplied code. When the program executes a call via the function pointer, the attacker's code is executed instead of the originally intended code. This can be an effective alternative to replacing the saved return value address in situations in which a function pointer is a local variable (or a field in a complex data type such as a C/C++ `struct` or `class`). In `f3` (shown in Figure 2), for example, assuming the local function variable `f` is laid on the stack after `buff`, the attacker can use the buffer overrun to modify `f`. If the attacker sets `f` to `buff`'s address (or a trampoline to `buff`), the call to `f` will transfer control to code that has been injected into `buff`.

Function-pointer clobbering combines effectively with arc injection. In `f2a`, for example, the attacker might choose to overwrite `f` with a location in the `system` function.

Function-pointer clobbering also combines very effectively with pointer subterfuge. A popular exploit on Unix and Linux systems uses an arbitrary pointer write to clobber pointers in the `fnlist` structure (typically manipulated by the `atexit` call); these functions are invoked as a process exits.

Although the earlier example is in terms of a stack buffer overrun, function-pointer clobbering can also be used on overruns embedded in heap structs or objects that contain embedded function pointers.

Function-pointer clobbering is an especially useful technique when the program being attacked uses some form of mitigation technique that prevents modification of the saved return address (for example, if a mechanism such as StackGuard[9] protects it). Because the saved return address is not overwritten, these mitigations do not prevent function-pointer clobbering exploits.

```
(a)
void f2a(void * arg, size_t len) {
  char buff[100];
  void (*f)() = ...;
  memcpy(buff, arg, len); /* buffer overrun! */
  f();
  /* ... */
  return;
}

(b)
void f2b(void * arg, size_t len) {
  char buff[100];
  long val = ...;
  long *ptr = ...;
  extern void (*f)();

  memcpy(buff, arg, len); /* buffer overrun! */
  *ptr = val;
  f();
  /* ... */
  return;
}
```

Figure 2. Pointer variables let the attacker use different exploit techniques. (a) A buffer overrun where the attacker can modify a function pointer. (b) A buffer overrun where an attacker can modify a data pointer, thus indirectly modifying a function pointer.

Data-pointer modification

If an address is used as a target for a subsequent assignment, controlling the address lets the attacker modify other memory locations, a technique known as an *arbitrary memory write*. In `f4`, for example, the buffer overrun of `buff` also modifies `ptr` and `val` values; this means that the attacker can use the assignment `*ptr = val` to set any four bytes of memory to a value of his or her choice. Data-pointer modification is thus useful as a building block in more complex exploits. (Although this example is in terms of a stack buffer overrun, an exploit can also use data-pointer modification on overruns embedded in heap structs or objects that also contain embedded pointers.)

A common use of data-pointer modification is in combination with function-pointer clobbering. In `f4` in Figure 2, because variable `f` is not a local variable, there is no way to clobber it as part of the buffer overrun. However, if the attacker knows `f`'s address, then the arbitrary pointer write can be used to change `f`'s value, thus leading

```
extern int global_magic_number;
void vulnerable(char *input, size_t max_length) {
     char buff[MAX_SIZE];
     int *ptr;
     int i;
     int stack_cookie;

P0:  stack_cookie = global_magic_number;
P1:  memcpy(buff, input, max_length); // buffer overrun!
   ...
P2:  *ptr = i;
P3:  if (stack_cookie != global_magic_number)
        // buffer overrun has occurred!
        exit();
P4:  return;
}
```

Figure 3. A simplified example of vulnerable code with compiler-inserted checks for buffer overruns. Pointer subterfuge attacks can be used to defeat compiler-inserted checks for buffer overruns, as this simplified example of vulnerable code illustrates.

to the same effect as a function-pointer clobber.

Another use of data-pointer modification is to modify some location used in future security-critical decisions. An exploit published by Litchfield for the MS03-026 vulnerability[7] is an excellent example of how pointer subterfuge can combine with a traditional stack-smashing approach.

Figure 3 presents a simplified version of a vulnerable program's code. The `stack_cookie` and `global_magic_number` variables are explicit (albeit simplified) representations of some additional checks introduced by the Microsoft Visual C++ 7.0 compiler. For the purposes of this section, it is enough to note that the goal of these code fragments is to prevent execution from reaching program point `P` if a buffer overrun has modified the saved return address, which on an x86 machine is located on the stack directly after the `stack_cookie` variable.

By taking advantage of the buffer overrun at `P1`, Litchfield's exploit first performs the following primitive operations:

- stores a sequence of instructions in the local variable `buff`;
- modifies the pointer variable `ptr` to point to the global variable `global_magic_number`;
- sets `i` to a new value `val`;
- sets `stack_cookie` to the same value `val`; and
- modifies the saved return address to contain the address of a trampoline that will indirectly transfer execution to `buff`.

When execution reaches `P2`, the assignment modifies the value in the global variable `global_magic_number`, setting it to `val`. As a result, the test at `P3` fails to detect the buffer overrun. Execution thus reaches `P4`, where the return instruction results in a control-flow transfer to the saved return value; the trampoline in turn transfers control to the instructions that have been stored in `buff`, and the exploit is successful.

As this example illustrates, data-pointer modification is an especially useful technique when the program being attacked uses a mitigation technique that prevents more straightforward exploits.

Exception-handler hijacking

Several variations of exploit techniques target the Microsoft Windows Structured Exception Handling (SEH) mechanism. When an exception (such as an access violation) is generated, Windows examines a linked list of exception handlers (typically registered by a program as it starts up) and invokes one (or more) of them via a function pointer stored in the list entry. Because the list entries are stored on the stack, it is possible to replace the exception-handler function pointer via buffer overflow (a standard example of function-pointer clobbering), thus allowing an attacker to transfer control to an arbitrary location—typically a trampoline to code injected by the attacker. Versions of Windows starting with Windows Server 2003 perform some validity checking of the exception handlers that limit the feasibility of this straightforward attack.

An alternative to clobbering an individual function pointer is to replace the field of the thread environment block (a per-thread data structure maintained by the Windows operating system) that points to the list of registered exception handlers. The attacker simply needs to "mock up" an apparently valid list entry as part of the payload and, using an arbitrary pointer write, modify the "first exception handler" field. Although recent versions of Windows do some validity checking for the list entries, Litchfield has demonstrated successful exploits in many of these cases.

As this discussion implies, the SEH mechanism has been repeatedly revised in response to these attacks; all currently published vulnerabilities are expected to be fixed in Windows XP Service Pack 2. However, published exploits continue to be feasible on older versions of the operating system, and it is, of course, possible that new exploits will be discovered for SEH's latest version.

While SEH is a Windows-specific concept, there could be analogous techniques on other operating systems. On Linux, for example, replacing an entry in the `fnlist` (discussed in the "Function-pointer clobbering" section) has some strong similarities to this approach.

Exception-handler hijacking is an especially useful technique when the program being attacked uses a mitigation technique that prevents more straightforward exploits of stack buffer overruns. In many cases, by triggering an unexpected exception, the attacker might be able to bypass the mitigation.

VPTR smashing

Most C++ compilers implement virtual functions via a *virtual function table* (VTBL) associated with each class; the VTBL is an array of function pointers that is used at runtime to implement dynamic dispatch. Individual objects, in turn, point to the appropriate VTBL via a *virtual pointer* (VPTR) stored as part of the object's header. Replacing an object's VPTR with a pointer to an attacker-supplied VTBL (which we will refer to as a "mock VTBL") allows the attacker to transfer control when the next virtual function is invoked.

VPTR smashing attacks can apply both to stack buffer overruns and heap buffer overruns, and can be used in conjunction with code injection or arc injection; Figure 4 illustrates this with a heap buffer overrun and a code injection attack.

In Figure 4, assume that the class `C` has a virtual function `vf`; further assume that the object for `ptr` is allocated directly after `buff`. In this case, the attacker has presumably transferred the payload (mock VTBL and code to be executed) in an earlier operation, and uses the buffer overrun to modify `ptr`'s VTBL. For sufficiently large stack buffer overruns, the mock VTBL and injected code can also be provided as part of the buffer overrun operation.

For a heap buffer overrun, it initially seems that the attacker must guess the type of object allocated directly after the buffer that has been overrun; while this is trivial in the example in Figure 4 it could be far more difficult in real-world situations. However, overwriting the VPTR to point to an attacker-supplied mock VTBL (in which each entry points to the attacker's injected code) removes this requirement.

VPTR smashing has not yet been used widely in practice, but it is a potentially useful technique when the program being attacked uses a mitigation technique that prevents heap smashing.

Heap smashing

Until very recently, experts believed that only stack buffers were vulnerable to exploitation. Various pointer-subterfuge exploits started to challenge this assumption, and led to the development of heap-specific attacks that have since become known as heap smashing.

```
void f4(void * arg, size_t len) {
  char *buff = new char[100];
  C *ptr = new C;

  memcpy(buff, arg, len); /* buffer overrun! */
  ptr->vf(); // call to a virtual function
  return;
}
```

Figure 4. A buffer overrun in C++ code, exploitable both by VPTR smashing and heap smashing.

The key insight behind heap smashing is to exploit the implementation of the dynamic memory allocator by violating some assumed invariants. (Published source code for the dynamic memory allocator makes explanation easier, but is not needed in practice, so these techniques apply equally well on closed-source systems.) Many allocators, for example, keep headers for each heap block chained together in doubly linked lists of allocated and freed blocks, and update these during operations such as freeing a memory block. If there are three adjacent memory blocks *X*, *Y*, and *Z*, an overrun of a buffer in *X* that corrupts the pointers in *Y*'s header can thus lead to modification of an arbitrary memory location when *X*, *Y*, or *Z* is freed. In many cases, the attacker can also control the value being put into that location, thus accomplishing an arbitrary memory write, which leads to the exploitation possibilities discussed in the "Data-pointer modification" section. In practice, heap smashing is thus typically coupled with function-pointer clobbering.

Three factors complicate heap-smashing exploits. Most obviously, the attacker typically does not know the heap block's location ahead of time, and standard trampolining approaches are typically not effective. In many cases, it is somewhat difficult to predict when the heap-free operation will occur, which could mean that the payload is no longer available at the time that the call via the clobbered function pointer occurs. Finally, in some situations (especially with multithreaded programs), it is difficult to predict whether the next block has been allocated at the time the overrun occurs.

Surprisingly enough, however, these are no longer significant roadblocks for exploitation of many heap buffer overruns. Attackers typically work around them by transferring the payload to an easy-to-find location as part of a separate operation (a technique first developed as a stack-smashing enhancement). There are typically enough such locations available that attackers can choose a location that will still be available at the time the call occurs. For cases where it is difficult to predict the next block, attackers can attempt to influence the size and

Nontraditional literature on buffer overruns

Although very little work has been published on exploitation in traditional conferences and journals, there is a lively parallel world—where the work is often of surprisingly high quality. This important resource is often left untapped by security researchers (although the references cited in the main article text in turn contain some useful references to this nontraditional literature).

The exploit techniques discussed in the main article come from four major threads of nontraditional literature, with a fair amount of crossover between them: Web sites and advisories from security companies and individual researchers; mailing lists, most notably the Security Focus VulnWatch and VulnDev mailing lists; hacker conferences such as Black Hat (www.blackhat.org); and *Phrack* magazine (www.phrack.org).

Stack smashing

AlephOne's 1996 *Smashing the Stack for Fun and Profit* (in Phrack 49 at www.phrack.org/show.php?p=49&a=14) and DilDog's *The Tao of Windows Buffer Overruns* (www.cultdeadcow.com/cDc_files/cDc-351/) are classic introductions to stack-smashing techniques and trampolining. Murat Balaban (www.enderunix.org/docs/eng/bof-eng.txt) first described the technique of storing the executable code in an environment variable. eEye's *Blaster Worm Analysis* (www.eeye.com/html/Research/Advisories/AL20030811.html) discusses the Blaster worm in detail.

Arc injection

Return-into-libc attacks were pioneered by Solar Designer in 1997 (www.securityfocus.com/archive/1/7480) and refined by Rafal Wojtczuk (1998's *Defeating Solar Designer's Non-executable Stack Patch* at www.insecure.org/sploits/non-executable.stack.problems.html, and 2001's *The Advanced return-into-lib(c) Exploits* in Phrack 58 at www.phrack.org/show.php?p=58&a=4). K-otic describes a return-into-libc exploit for MS03-026 at www.k-otik.com/exploits/11.07.rpcexec.c.php.

Pointer subterfuge

Pointer subterfuge attacks were developed largely in response to the introduction of stack canary checking in StackGuard and other products. Matt Conover's 1999 paper on heap exploitation (www.w00w00.org/files/articles/heaptut.txt) discusses this and has several examples of pointer subterfuge attacks; he cites Tim Newsham's earlier mail suggesting this approach. Bulba and Kil3r's *Bypassing Stackguard and Stackshield* (Phrack 56 at www.phrack.org/show.php?p=56&a=5) and Gerardo Richarte's similarly titled *Bypassing the Stackguard and Stackshield Protection* (www2.corest.com/common/showdoc.php?idx=242&idxseccion=11) are two early examples here. David Litchfield discusses exception-handler hijacking in his 2003 paper on *Variations in Exploit Methods Between Linux and Windows* (www.nextgenss.com/papers/exploitvariation.pdf); his *Defeating the Stack Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server* (www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf) discusses both pointer subterfuge in general and exception handling hijacking in particular. Rix described *Smashing C++ VPTRs* in Phrack 56 (at www.phrack.org/show.php?p=56&a=8).

Heap smashing

Conover's 1999 paper (mentioned above) first described heap exploitation techniques, although he noted "... heap-based overflows are not new." A pair of articles in Phrack 57 (Michel Kaempf's *Vudo Malloc Tricks* at www.phrack.org/show.php?p=57&a=8 and Anonymous' *Once Upon a Free ...* at www.phrack.org/show.php?p=57&a=9) introduced heap smashing in mid 2001. Halvar Flake's *Third Generation Exploits* (www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt) first applied this to Windows; Litchfield's *Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP* (www.nextgenss.com/papers/non-stack-bo-windows.pdf) expands on these concepts.

JP's *Advanced Doug Lea's Malloc Exploits* (Phrack 61 at www.phrack.org/show.php?p=61&a=6) elegantly analyzes heap-based exploits in terms of primitive operations and covers several techniques for extracting information from the target program to make exploits more reliable. Pheonlit's description of exploiting a Cisco IOS heap overrun (www.phenoelit.de/ultimaratio/index.html) shows how attackers can work around partial defenses. Frédéric Pierrot and Peter Szor of the Symantec Security Response Center analyzed the Slapper worm's use of the heap-smashing technique (securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf). Dave Aitel's excellent two-part series *Exploiting the MSRPC Heap Overflow* (www.immunitysec.com/papers/msrpcheap.pdf) gives a good feel for now hackers develop exploits in practice, and also illustrates the technique of providing the "payload" in an operation distinct from the buffer overrun itself.

order of heap operations to position the heap block of interest at a known location and to disambiguate the following blocks' behavior.

As of early 2004, the clear explanations in *The Shellcoder's Handbook*[6] and elsewhere imply that heap buffer overrun exploitation is almost as cookbook of a process as stack buffer exploitation. Just as the original stack-smashing technique has been repeatedly extended, researchers such as Matt Conover and Oded Horovitz are currently investigating new exploit approaches that build on heap smashing.[8]

Given the continued existence of large amounts of C and C++ code in system-critical code (operating

systems, databases, mail and Web servers, and so on), buffer overruns are likely to continue to be an important source of potential vulnerabilities. Security researchers continue to devise new mitigation approaches to preventing exploitation of these vulnerabilities, as well as to investigate promising combinations[10]; however, attackers have proven equally adept at inventing new exploit techniques to defeat promising mitigations.

An important point about the new exploitation techniques we described in this article is that they invalidate traditional assumptions about buffer overrun exploits.

- Arc injection invalidates the assumption that all exploits rely on injecting code.
- Pointer subterfuge invalidates the assumption that all exploits rely on overwriting the saved return address.
- Heap smashing and pointer subterfuge both invalidate the assumption that only stack buffers are vulnerable.

In addition to taking these specific techniques into account when analyzing potential solutions, researchers must also consider what assumptions the solutions make. Similarly, systems designers must consider exploit possibilities as they introduce new functionality in order to avoid introducing new avenues of exploitation (for example, C++ virtual functions, structured exception handling, or `atexit`). At least in the short term, the "arms race" appears likely to continue: attackers will find ways to invalidate assumptions, and thus create new exploit techniques. □

References

1. *CERT Advisory CA-2002-27 Apache/mod_ssl Worm*, CERT Coordination Ctr., 2002; www.cert.org/advisories/CA-2002-27.html.
2. *Buffer Overrun in RPC Interface Could Allow Code Execution*, Microsoft Security Bulletin MS03-0262003; www.microsoft.com/technet/security/bulletin/MS03-026.mspx.
3. C. Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conf. and Expo* (DISCEX '00), 2000; www.immunix.com/pdfs/discex00.pdf
4. J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. 10th Network and Distributed System Security Symp.* (NDSS '03), 2003; www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf.
5. G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.
6. J. Koziol et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, John Wiley & Sons, 2004.
7. D. Litchfield, *Defeating the Stack Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*, 2003; www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf.
8. M. Conover and O. Horovitz, "Reliable Windows Heap Exploits," *Proc. CanSecWest*, 2004; http://cansecwest.com/csw04/csw04-Oded+Connover.ppt.
9. C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer overrun attacks," *Proc. Usenix Security Symp.*, Usenix Assoc., 1998; www.immunix.com/pdfs/usenixsc98.pdf.
10. J. Pincus and B. Baker, *Mitigations for Low-Level Coding Vulnerabilities: Incomparability and Mitigations*; http://research.microsoft.com/users/jpincus/mitigations.pdf.

Jonathan Pincus *is a senior researcher at Microsoft Research. His research interests include security, privacy, and reliability of software-based systems. He received his MS from the University of California at Berkeley, and his AB from Harvard University. Contact him at jpincus@microsoft.com.*

Brandon Baker *is a security development engineer and red-team lead in the Microsoft Windows Security group. His research interests include security architecture and penetration testing. He received his BSc in computer science from Texas A&M University. Contact him at babaker@microsoft.com.*