Launching an executable from a memory buffer

The ability to execute a program directly from memory greatly enhances its stealth. The first such technique was devised by Gary Nebbett in December 2000. It is referred to as Nebbett's Shuttle. This technique was windows specific and utilizes the Win32 API as well as internal windows functions to accomplish its purpose.

The essence of Nebbett's approach was to launch a process in a suspended state. Once in this suspended state its process memory could be manipulated and overwritten with a new executable. Execution of the process is then resumed.

Covertly launching an executable using Nebbett's Shuttle allows the malicious process to appear as the process which it was started as. For example, if one were to use this technique with notepad.exe being the process which is started in a suspended state, then overwritten with sol.exe, a game of solitaire would be presented to the user. When the user views their task manager it will appear as though the notepad program is running.

Injecting code in this fashion is also a means to bypass security mechanisms such as host-based firewalls. Host-based firewalls often have a list of allowed programs which are the only programs it will allow to communicate in certain fashions. If one were to use Nebbett's shuttle to start up a program which is included in this exception list and inject malware into it, that malware will have effectively bypassed the host-based firewall mechanism.

These techniques are of particular note to the field of Anti-Forensics however, because it allows for the execution of programs which do not need to be present on disk media. A malware launcher can be crafter which downloads a program from the internet and executes it without leaving any disk residue of this action taking place. With no disk residue a forensic examiner will also be unable to capture the malicious binary for reverse engineering.

A UNIX equivalent to Nebbett's shuttle is a tricky problem. Anti-Forensic researcher "the grugq" has devised two techniques to address this issue. His first technique is a simple user land implementation of the execve() function call. This allows a UNIX process to load and execute an ELF binary image from a memory buffer.

The grugq's second technique involved a shellcode loader which could be injected into a vulnerable process. This shellcode loader would then download what he named an lxobject. An lxobject is a package containing an ELF binary, associated stack and context data, and shellcode to self-launch the executable in the current process space. Once the lxobject was downloaded it simply passed execution to the shellcode portion of the lxobject and the process would be overtaken by the malicious executable image.

The approach to executing a process directly from memory under unix which I describe for the first time in this presentation will take the following approach. First, a program which uses debugger techniques will launch or attach to a process. If the process is launched by the debugger then a breakpoint will be placed at the entry point. Otherwise, if the debugger attaches to a running process then a breakpoint is placed at the currently executing instruction. When the breakpoint is triggered, control flow of the program belongs to the debugger.

The first step taken against the suspended process is to inject a small piece of shellcode at the currently executing memory location whose purpose it is to allocate a small buffer outside of the location occupied by the executables memory image. The final instruction in this shellcode will be a software interrupt (int 3). When the interrupt instruction is executed control will return to the debugger.

The next action performed by the debugger is the injection of a second piece of shellcode. This time the shellcode is inserted into the memory buffer which was just created by the first piece of shellcode. The job of the second piece of shellcode is first to unmap the executable image from memory. The reason for this is so that we can place our new executable in its favored location and eliminate some compatibility issues associated with this form of injection. Next the shellcode allocates the amount of memory needed to contain the new executable memory image. It then grows the heap if needed, according to what is required by the new executable. Once again, the finial instruction in the shellcode is the software interrupt instruction.

After control returns to the debugger it must populate the target process' memory with the new executable image. It will also populate the stack with the appropriate data for the new process and set registers to the values appropriate for a new process. At this point the debugging process may detach itself from the target if necessary as execution is resumed on manipulated process.

A portable mechanism exists for performing debugging operations on most UNIX variants. This interface, the ptrace() system call, provides capabilities to attach to a remote process, debug a child process, read and write process memory, manipulate registers and alter signal handlers in the target process. Using the approach of the ptrace() system call to manipulate processes it is possible to achieve other process injection techniques beyond what we have just discussed.

References

1. Gary Nebbett, http://groups.google.com/group/comp.os.ms-windows.programmer.win32/msg/3fa4e10562db1d03?output=gplain
2. the grugq, http://archives.neohapsis.com/archives/fulldisclosure/2003-q4/3897.html
3. the grugq, http://www.phrack.org/issues.html?issue=63&id=11