Injecting Shellcode Into Running VMware Guests

This document describes at a high level how to escape the virtual boundaries that isolate a host system from its guest operating system(s). There is no discussion of how to initially gain control over the host – this is left up to the exploit designer. Before reading on, please realize that no specific vulnerability in the VMware code base is being exploited to accomplish this task. As such, a vendor patch or fix should not be expected. Additionally, due to the potential impact that this information could have on virtual server farms, many sections are left intentionally vague.

This project began when a friend mentioned the challenge on an otherwise not-so-happenin' Friday morning and offered a plane ticket as inspiration. Unfortunately, flight details are still controversial; however the results of the challenge are pretty solid. My research was conducted on an x86 architecture running Windows XP as both the host and guest. I have not extended this testing to other platforms or to environments running virtualization software other than VMware. Theoretically, the success rate would be quite high based on principle alone.

In order to understand this technique, one must become familiar with the basic structure of a guest's memory and its interaction with the host in order to access the memory. A process on the host, namely vmware-vmx.exe, maps n bytes of data into its own virtual address space, where n is equal to the size of the guest's RAM. In order to maintain state for the guest between pauses and snapshots, this data is flushed and saved to a file on disk. For integrity, the vmware-vmx.exe process maintains a read/write lock on the file while the guest is active.

The first challenge was gaining access to this data without disturbing the stability of the guest. If one attempts to release the locking mechanism, then it would need to be re-applied afterward. Even more extreme, if the process' handle to the file is closed then it would need to be re-created. Neither of these options describe a reliable way to access the data, however it remains a very attractive resource. If an exploit designer can exercise read and write operations on the resource, then the guest system literally becomes a playground. Kernel structures can be owned, functions can be hooked, and any old pointers can be overwritten…not to mention any un-encrypted data in RAM can be exposed.

The most direct way to access the data from an external perspective is to, well, change perspectives. Instead of trying to overcome the locks, if code can be executed within the process space of the vmware-vmx.exe program, then accessibility problems are alleviated. This was the selected approach that later evolved into a DLL which is injected into vmware-vmx.exe by using a few Windows API functions such as CreateToolHelp32Snapshot(), VirtualAllocEx(), WriteProcessMemory(), and CreateRemoteThread(). Once again, identifying a vulnerability on the host system which allows for arbitrary code execution is not a topic of this document.

At this point, a function exported by the DLL can crawl the virtual address space that belongs to the vmware-vmx.exe process looking for the particular memory-mapped file that corresponds to the guest's RAM image. A unique file extension makes this procedure very accurate. The base address of the mapped file can be located within a fraction of a second, however all n bytes are practically guaranteed to be non-contiguous (at least I never saw the whole n bytes allocated sequentially). This needs to be taken into consideration when crawling the address ranges, because accessing invalid locations may cause exceptions and even more so – it's a huge waste of time. An improved algorithm for jumping over invalid memory ranges reduced the code's run time from 5-6 minutes down to about 20 seconds for a 256MB mapped file.

The point is to find a sequence of bytes within the RAM image that represents an exported function from the guest's kernel32 library, and replace it with shellcode. As mentioned before, this is only one of many possibilities, but it is the one I chose for the project. A signature was created from the first 16 bytes of an export and several verification rounds followed in order to make sure that the signature was free of false-positives and false-negatives. For testing purposes, my signature came from a rarely used function, but a more aggressive and efficient attack would select a popular one to reduce the amount of time one has to wait for any process on the guest to call the function.

This technique has obvious caveats such as overwriting the legitimate function's code. A cleaner approach would be to replace the first few bytes with a jump or call to previously staged shellcode elsewhere in the RAM image and then return to the legitimate function. It also makes an assumption based on the guest's version of kernel32.dll and in general, assumptions are bad. So in conclusion, there is room for design improvement, but the existing method surpassed my expectations for both efficiency and reliability based on the amount of time applied to the project.

A proof-of-concept video of this technique in action is accessible from http://www.mnin.org/advisories/vmshell/vmshell.html.