An Objective Analysis of the Lockdown Protection System for Battle.net

12/2007

Skywing
skywing@valhallalegends.com

Abstract

Near the end of 2006, Blizzard deployed the first major update to the version check and client software authentication system used to verify the authenticity of clients connecting to Battle.net using the binary game client protocol. This system had been in use since just after the release of the original Diablo game and the public launch of Battle.net. The new authentication module (Lockdown) introduced a variety of mechanisms designed to raise the bar with respect to spoofing a game client when logging on to Battle.net. In addition, the new authentication module also introduced run-time integrity checks of client binaries in memory. This is meant to provide simple detection of many client modifications (often labeled "hacks") that patch game code in-memory in order to modify game behavior. The Lockdown authentication module also introduced some anti-debugging techniques that are designed to make it more difficult to reverse engineer the module. In addition, several checks that are designed to make it difficult to simply load and run the Blizzard Lockdown module from the context of an unauthorized, non-Blizzard-game process. After all, if an attacker can simply load and run the Lockdown module in his or her own process, it becomes trivially easy to spoof the game client logon process, or to allow a modified game client to log on to Battle.net successfully. However, like any protection mechanism, the new Lockdown module is not without its flaws, some of which are discussed in detail in this paper.

1 Introduction

The Lockdown module is a part of several schemes that attempt to make it difficult to connect to Battle.net with a client that is not a "genuine" Blizzard game. For the purposes of this paper, the author considers both modified/"hacked" Blizzard game clients, and third-party client software, known as "emubots", as examples of Battle.net clients that are not genuine Blizzard games. The Battle.net protocol also incorporates a number of schemes (such as a proprietary mechanism for presenting a valid CD-Key for inspection by Battle.net, and a non-standard derivative of the SRP password exchange protocol for account logon) that by virtue of being obscure and undocumented make it non-trivial for an outsider to successfully log a non-genuine client on to Battle.net.

Prior to the launch of the Lockdown module, a different system took its place and filled the role of validating client software versions. The previous system was resistant to replay attacks (caveat: a relatively small pool of challenge response values maintained by servers makes it possible to use replay attacks after observing a large number of successful logon attempts) by virtue of the use of a dynamically-supplied checksum formula that is sent to clients (a challenge, in effect). This formula was then interpreted by the predecessor to the Lockdown module, otherwise known as the "ver" or "ix86ver" module, and used to create a one-way hash of several key game client binaries. The result response would then be sent back to the game server for verification, with an invalid response resulting in the client being denied access to Battle.net.

While the "ver" module provides some inherent resistance to some types of non-genuine clients (such as those that modify Blizzard game binaries on disk), it does little to stop in-memory modifications to Blizzard game clients. Additionally, there is very little to stop an attacker from creating their own client software (an "emubot") that implements the "ver" module's checksum scheme, either by calling "ver" directly or through the use of a third-party, reverse-engineered implementation of the algorithm implemented in the "ver" module. It should be noted that there exists one basic protection against third party software calling the "ver" module directly; the "ver" series of modules are designed to always run part of the version check hash on the caller process image (as returned by the Win32 API GetModuleFileNameA). This poses a minor annoyance for third party programs. In order to bypass this protection, however, one need only hook GetModuleFileNameA and fake the result returned to the "ver" module.

Given the existing "ver" module's capabilities, the Lockdown module represents a major step forward in the vein of assuring that only genuine Blizzard client software can log on to Battle.net as a game client. The Lockdown module is a first in many respects for Blizzard with respect to releasing code that actively attempts to thwart analysis via a debugger (and actively attempts to resist being called in a foreign process with non-trivial mechanisms).

Despite the work put into the Lockdown module, however, it has proven perhaps less effective than originally hoped (though the author cannot state the definitive expectations for the Lockdown module, it can be assumed that a "hacking life" of more than several days was an objective of the Lockdown module). This paper discusses the various major protection systems embedded into the Lockdown module and associated authentication system, potential attacks against them, and technical counters to these attacks that Blizzard could take in a future release of a new version check/authentication module.

Part of the problem the developers of the Lockdown module faced relates to constraints on the environment in which the module operates. The author has derived the following constraints currently in place for the module:

1. The server portion of the authentication system is likely static and does not generate challenge/response values in real time. Instead, a pool of possible values appear to be pregenerated and configured on the server.

2. The module needs to work on all operating systems supported by all Blizzard games, which spans the gamut from Windows 9x to Windows Vista x64. Note that there are provisions for different architectures, such as Mac OS, to use a different system than Windows architectures.

3. The module needs to work on all versions of all Blizzard Battle.net games, including previous versions. This is due to the fact that the module plays an integral part in Battle.net's software version control system, and thus is used on old clients before they can be upgraded.

4. Legitimate users should not see a high incidence of false positives, and it is not desirable for false positives to result in automated permanent action against legitimate users (such as account closure).

As an aside, in the author's opinion, the version check and authentication system is not intended as a copy protection system for Battle.net, as it does nothing to discourge additional copies of genuine Blizzard game software from being used on Battle.net. In essence, the version check and authentication system is a system that is designed to ensure that *only* copies of the genuine Blizzard game software can log on to Battle.net. Copy protection measures on Battle.net are provided through the CD-Key feature, wherein the server requires that a user has a valid (and unique) CD-Key (for applicable products).

2 Protection Schemes of the Lockdown Module

As a stark contrast to the old "ver" module, the Lockdown module includes a number of active defense mechanisms designed to significantly strengthen the module's resistance to attack (including either analysis or being tricked into providing a "good" response to a challenge to an untrusted process).

The protection schemes in the Lockdown module can be broken up into several categories:

1. Mechanisms to thwart analysis of the Lockdown module itself and the secret algorithm it implements (anti-debugging/anti-reverse-engineering).

2. Mechanisms to thwart the successful use of Lockdown in a hostile process to generate a "good" response to a challenge from Battle.net (anti-emubot, and by extension anti-hack, where

"anti-hack" denotes a counter to modifications of an otherwise genuine Blizzard game client).

3. Mechanisms to thwart modifications to an otherwise-genuine Blizzard game client that is attempting to log on to Battle.net (anti-hack).

In addition, the Lockdown module is also responsible for implementing a reasonable facsimile of the original function of the "ver" module; that is, to provide a way to authoritatively validate the version of a genuine Blizzard game client, for means of software version control (e.g. the deployment of the correct software updates/patches to old versions of genuine Blizzard game clients connecting to Battle.net).

In this vein, the following protection schemes are present in the Lockdown module and associated authentication system:

2.1 Clearing the Processor Debug Registers

The x86 family of processors includes a set of special registers that are designed to assist in the debugging of programs. These registers allow a user to cause the processor to stop when a particular memory location is accessed, as an instruction fetch, as a data read, or as a data write. This debugging facility allows a user (debugger) to set up to four different virtual addresses that will trap execution when referenced in a particular way. The use of these debug registers to set traps on specific locations is sometimes known as setting a hardware breakpoint, as the processor's dedicated debugging support (in-hardware) is being utilized.

Due to their obvious utility to anyone attempting to analyze or reverse engineer the Lockdown module, the module actively attempts to disable this debugging aid by explicitly zeroing the contents of the key debug registers in the context of the thread executing the Lockdown module's version check call, CheckRevision. All the requisite debug registers are cleared immediately after the call to the CheckRevision routine in the Lockdown module is made.

This protection mechanism constitutes an anti-debugging scheme.

2.2 Memory Checksum Performed on the Lockdown Module

The Lockdown module, contrary to the behavior of its predecessor, implements a checksum of several key game executable files in-memory instead of on-disk. In addition to the checksum over certain game executables, the Lockdown module includes itself in the list of modules to be checksumed. This provides several immediate benefits:

1. Attempts to set conventional software breakpoints on routines inside the Lockdown module will distort the result of the operation, frustrating reverse engineering attempts. This is due to the fact that so-called software breakpoints are implemented by patching the instruction at the target location with a special instruction (typically 'int 3') that causes the processor to break into the debugger. The alteration to the module's executable code in memory causes the checksum to be distorted, as the 'int 3' opcode is checksumed instead of the original opcode.

2. Attempts to bypass other protection mechanisms in the Lockdown module are made more difficult, as an untrusted process that is attempting to cause the Lockdown module to produce correct results via patching out certain other protection mechanisms will, simply by virtue of altering Lockdown code in-memory, inadvertently alter the end result of the checksum operation. The success of this aspect of the memory checksum protection is related to the fact that the Lockdown module attempts to disable hardware breakpoints as well. These two protection mechanisms thus complement eachother in a strong fashion, such that a naive attempt to compromise one of the protection schemes would

usually be detected by the other scheme. In effect, the result is a rudimentary "defense in depth" approach to software protection schemes that is the hallmark of most relatively successful protection schemes.

3. The inclusion of the version check module itself in the result of the output of the checksum is entirely new to the version check and client authentication system, and as such poses an additional, unexpected "speed bump" to persons attempting to reimplement the Lockdown algorithm in their own code.

This protection mechanism has characteristics of both an anti-debugging, anti-hack, and anti-emubot system.

2.3 Hardcoding of Module Base Addresses

As mentioned previously, the Lockdown module now implements a checksum over game executables in-memory instead of on-disk. Taking advantage of this change, the Lockdown module can hardcode the base address of the main process executable at the default address of 0x00400000. This is safe because no Blizzard game executable includes base relocation information, and as a result will never change from this base address.

By virtue of hardcoding this address, it becomes more difficult for an untrusted process to successfully call the Lockdown module. Unless the programmer is particularly clever, he or she may not notice that the Lockdown module is not actually performing a checksum over the main executable for the desired Blizzard game, but instead the main executable of the untrusted process (the default address for executables in the Microsoft linker program is the same 0x00400000 value used in Blizzard's main executables comprising their game clients).

While it is possible to change the base address of a program at link-time, which could be done by a third-party process in an attempt to make it possible to map the desired Blizzard main executable at the 0x00400000 address, it is difficult to pull this off under Windows NT. This is because the 0x00400000 address is low in the address space, and the default behavior of the kernel's memory manager is to find new addresses for memory allocations starting from the bottom of the address space. This means that in virtually all cases, a virgin Win32 process will already have an allocation (usually one of the shared sections used for communication with CSRSS in the author's experience) that is overlapping the address range required by the Lockdown module for the main executable of the Blizzard game for which a challenge response is being computed. While it is possible to change this behavior in the Windows NT memory manager and cause allocations to start at the top of the address space and search downwards, this is not the default configuration and is also a relatively not-well-known kernel option. The fact that all users would need to be reconfigured to change the default allocation search preference for an untrusted process to typically successfully map the desired Blizzard game executable makes this approach relatively painful for a would-be attacker.

The Lockdown module also ensures that the return value of the GetModuleHandleA(0) Win32 API corresponds to 0x00400000, indicating that the main process image is based at 0x00400000 as far as the loader is concerned. The restriction on the base address of the game main executable module has the unfortunate side effect that it will not be possible to take advantage of Windows Vista's ASLR attack surface reduction capabilities, negatively impacting the resistance of Blizzard games to certain classes of exploitation that might impact the security of users.

This protection mechanism is primarily considered to be an anti-emubot scheme, as it is designed to guard against an untrusted process from succcessfully calling the Lockdown module.

2.4 Video Memory Checksum

Another previously nonexistant component to the version check algorithm that is introduced by the Lockdown module is a checksum over the video memory of the process calling the Lockdown module. At the point in time where the module is invoked by the Blizzard game, the portion of video memory checksummed should correspond to part of the "Battle.net" banner in the log on screen for the Blizzard game. The Lockdown module is currently only implemented for so-called "legacy" game clients, otherwise known as clients that use Battle.snp and the Storm Network Provider system for multiplayer access. This includes all Battle.net-capable Blizzard games ranging from Diablo I to Starcraft and Warcraft II: BNE. Future games, such as Diablo II, are not supported by the Lockdown module.

This represents an additional non-trivial challenge to a would-be attacker. Although the contents of the video memory to be checksummed is static, the way that the Lockdown module retrieves the video memory pointers is through an obfuscated call to several internal Storm routines (SDrawSelectGdiSurface, SDrawLockSurface, and SDrawUnlockSurface) that rely on a non-trivial amount of internal state initialized by the Blizzard game during startup. This makes the use of the internal Storm routines unlikely to simply work "out of the box" in an untrusted process that has not gone to all the trouble to initialize the Storm graphics subsystem and draw the appropriate data on the Storm video surfaces.

This protection mechanism is primarily considered to be an anti-emubot scheme, as it is designed to guard against an untrusted process from succcessfully calling the Lockdown module.

2.5 Multiple Flavors of the Lockdown Module

The original "ver" module scheme pioneered a system wherein there were multiple downloadable flavors of the version check module to be used by a client. The Battle.net server sends the client a tuple of (version check module filename, checksum formula and initialization parameters, version check module timestamp) that is used in order to version (and download, if necessary) the latest copy of the version check module. This mechanism provides for the possibility that the Battle.net server could support multiple "flavors" of version check module that could be distributed to clients in order to increase the amount of work required by anyone seeking to reimplement the version check and authentication system.

The original "ver" module and associated authentication scheme in fact utilized such a scheme of multiple "ver" modules, and the Lockdown scheme expands upon this trend. In the original system, there were 8 possible modules to choose from; the Lockdown system, by contrast, expands this to a set of 20 possibilities. However, the version check modules in both systems are still very similar to one another. In both systems, each module has its own unique key (a 32-bit values in the "ver" system, and a 64-bit value in the Lockdown system) that is used to influence the result of the version check checksum (it should be noted that in the Lockdown system, the actual Lockdown module itself is in essence a second "key", as the added checksum over the module represents an additional adjustment to the final checksum result that changes with each Lockdown module). This single difference is disguised by other minor, superficial alterations to each module flavor; there are slight differences by which module base addresses are retrieved, for instance, and there are also other superficial differences that relate to differences like code being moved between functions or functions being re-arranged in the final binary in order to frustrate a simple "diff" of two Lockdown modules as being informative in revealing the functional differences between the said two modules.

This protection mechanism is perhaps best classed as an anti-analysis scheme, as it attempts to create more work for anyone attempting to reverse engineer the authentication system as a whole.

2.6 Authenticity Check Performed on Lockdown Module Caller

An additional new protection scheme introduced in the Lockdown module is a rudimentary check on the authenticity of the caller of the module's export, the CheckRevision routine. Specifically, the module attempts to ascertain whether the return address of the call to the CheckRevision routine points to a code location within the Battle.snp module. If the return pointer for the call to CheckRevision is not within the expected range, then an error is deliberately introduced into the checksum calculations, ultimately resulting in the result returned by the Lockdown module becoming invalidated.

3 Attacks (and Counter-Attacks) on the Lockdown System

Though the Lockdown module introduces a number of new defensive mechanisms that attempt to thwart would-be attackers, these systems are far from foolproof. There are a number of ways that these defensive systems could be attacked (or subverted) by a would-be attacker who wishes to pass the version and authentication check in the context of a non-genuine client for purposes of logging on to Battle.net. In addition, there are also a variety of different ways by which these proposed attacks could be thwarted in a future update to the version check and authentication system.

3.1 Interception of SetThreadContext

As previously described, the Lockdown modules attempt to disable the use of the processor's complement of debug registers in order to make it difficult to utilize so-called hardware breakpoints during the process of reverse engineering or analyzing a Lockdown module. This scheme is, at present, relatively easily compromised, however.

There are several possible attacks that could be used:

1. Hook the SetThreadContext API and block attempts to disable debug registers (programmatic).
2. Patch the import address table entry for SetThreadContext in the Lockdown module to point to a custom routine that does nothing (programmatic).
3. Patch the Lockdown module instruction code to not call SetThreadContext in the first place (programmatic). However, this is approach is considered to be generally untenable, due to the memory checksum protection scheme.
4. Set a conditional breakpoint on 'kernel32!SetThreadContext' that re-applies the hardware breakpoint" state after the call, or simply alters execution flow to immediately return (debugger).

Depending on whether the attacker wants to make programmatic alterations to the behavior of the Lockdown module via hardware breakpoints, or simply wishes to observe the behavior of the module in the debugger unperturbed, there are several options available.

The suggested counters include techniques such as the following:

1. Verify that the debug registers were really cleared. However, this could simply be patched out as well. More subtle would be to include the value of several debug registers in the checksum calculations, but this would also be fairly obvious to attackers due to the fact that debug registers cannot be directly accessed from user mode and require a call to Get/SetThreadContext, or the underlying NtGet/SetContextThread system calls.

2. Include additional calls to disable debug register usage in different locations within the Lockdown module. To be most effective, these would need to be inlined and use different means to set the debug register state. For example, one location could use a direct import, another could use a GetProcAddress dynamic import, a third could manually walk the EAT of kernel32 to find the address of SetThreadContext, and a fourth could make a call to NtSetContextThread in ntdll, and a fifth could disassemble the opcodes comprising NtSetContextThread, determine the system call ordinal, and make the system call directly (e.g. via 'int 2e'). The goal here is to add additional work and eliminate "single points of failure" from the perspective of an attacker seeking to disable the anti-debugging feature. Note that the direct system call approach will require additional work in order to function under Wow64 (e.g. x64 computers running native Windows x64).

3. Verify that all IAT entries corresponding to kernel32 actually point to the same module in-memory. This is risky, though, as in some cases (such as when the Microsoft application compatibility layer module is in use), these APIs may be legitimately detoured.

3.2 Use of Hardware Breakpoints

Assuming an attacker can compromise the anti-debugging protection scheme, then he or she is free to make clever use of hardware breakpoints to disable other protection systems (such as hardcoded base addresses of modules, checks on the authenticity of a CheckRevision caller, and soforth) by setting execute fetch breakpoints on choice code locations. Then, the attacker could simply alter the execution context when the breakpoints are hit, in order to bypass other protection mechanisms. For example, an attacker could set a read breakpoint on the hardcoded base address for the main process image inside the Lockdown module, and change the base address accordingly. The attacker would also have to patch GetModuleHandleA in order to complete this example attack.

Suggested counters to attacks based on hardware breakpoints include:

1. Validation of the vectored exception handler chain, which might be used to intercept STATUS_SINGLE_STEP exceptions when hardware breakpoints are hit. This is risky, as there are legitimate reasons for there to be "foreign" vectored exception handlers, however.

2. Checks to stop debuggers from attaching to the process, period. This is not considered to be a viable solution since there are a number of legitimate reasons for a debugger to be attached to a process, many of them which may be unknown completely to the end user (such as profilers, crash control and reporting systems, and other types of security software). Attempting to block debuggers may also prevent the normal operation of Windows Error Reporting or a preconfigured JIT debugger in the event of a game crash, depending on the implementation used. Ways of detecting debuggers include calls to IsDebuggerPresent, NtQueryInformationProcess(...ProcessDebugPort..), checks against NtCurrentPeb()-¿BeingDebugged, and soforth.

3. Duplication of checks (perhaps in slightly altered forms) throughout the execution of the checksum implementation. It is important for this duplication to be inline as much as possible in order to eliminate single points of failure that could be used to short-circuit protection schemes by an attacker.

4. Strengthening of the anti-debugging mechanism, as previously described.

3.3 Main Process Image Module Base Address Restriction

An attacker seeking to execute the Lockdown module in an untrusted process would need to bypass the

restrictions on the base address of the main process image. The most likely approach to this would be a combination attack, whereby the attacker would use something like a set of hardware breakpoints to alter the hardcoded restrictions on module base addresses, and import table or code patch style hooks on the GetModuleHandleA API in order to defeat the secondary check on the module base address for the main executable image.

Another approach would be to simply create the main executable image as a process, suspended, and then either create a new thread in the process or assume control of the initial thread in order to execute the Lockdown module. This gets the would-be attacker out of having to patch checks in the module, as there is currently no defense against this case implemented in the module.

In order to strengthen this protection mechanism, the following approaches could be taken:

1. Manually traverse the loaded module list (and examine the PEB) in order to validate that the main process image is really at 0x00400000. All of these mechanisms could be compromised, but checking each one creates additional work for an attacker.

2. Verify that the game has initialized itself to some extent. This would make the approach of creating the game process suspended more difficult. It would also otherwise make the use of the Lockdown module in an untrusted process more difficult without tricking the module into believing that it is running in an initialized game process. The scope of determining how the game is initialized is outside of this paper, although an approach similar to the current one based on a checksum of Storm video memory (though with more "redundancy", or an additional matrix of requirements for a legitimate game process).

3.4 Minor Functional Differences Between Lockdown Module Flavors

Presently, an attacker needs to implement all flavors of the Lockdown module in order to be assured of a successful connection to Battle.net. However, even with the 20 possibilities now available, this is still not difficult due to the minor functional differences between the different Lockdown flavors. Moreso, it is trivially possible to find the "magic" constants that constitute the only functional differences between each flavor of Lockdown.

In the author's tests, two pattern matches and a small 200-line C program were all that were necessary to programmatically identify all of the magical constants that represent the functional differences between each flavor of Lockdown module, in a completely automated fashion. In fact, the author would wager that it took more time to implement all 20 different flavors of Lockdown modules than it took to devise and implement a rudimentary pattern matching system to automagically discover all 20 magical constants from the set of 20 Lockdown module flavors. Clearly, this is not desirable from the standpoint of effort put in to the protection scheme vs difficulty in attacking it.

In order to address these weaknesses, the following steps could be implemented:

1. Implement true, major functional differences between Lockdown flavors. Instead of using a single constant value that is different between each flavor (probably a "`#define`" preprocessor constant), implement other, real functional differences. Otherwise, even with a number of different "non-functional" differences between module flavors, a pattern-matching system will be able to quickly locate the different constants for each module after a human attacker has discovered the constant for at least one module flavor.

2. Avoid using quick-to-substitute constants as the "meat" of the functional differences betwene flavors. While these are convenient from a develop-

ment perspective, they are also convenient from an attacker perspective. If a bit more time were spent from a development perspective, attackers could be made to do real analysis of each module separately in order to determine the actual functional differences, greatly increasing the amount of time that is required for an attacker to defeat this protection scheme.

3.5 Spoofed Return Address for CheckRevision Calls

Due to how the x86 architecture works, it is trivially easy to spoof the return address pointer for a procedure call. All that one must do is push the spoofed return address on the stack, and then immediately execute a direct jump to the target procedure (as opposed to a standard call).

As a result, it is fairly trivial to bypass this protection mechanism at run-time. One need only search for a 'ret' opcode in the code space of the Battle.snp module in memory, and use the technique described previously to simply "bounce" the call off of Battle.snp via the use of a spoofed return address. To the Lockdown module, the call will appear to originate from the context of Battle.snp, but in reality the call will immediately return from Battle.snp to the real caller in the untrusted process.

To counter this, the following could be attempted:

1. Verify two return addresses deep, although due to the nature of the x86 calling conventions (at least __stdcall and __fastcall, the two used by Blizzard code frequently), it is not guaranteed that four bytes past the return address will be a particularly meaningful value.

2. Verify that the return address does not point directly to a 'ret', 'jmp', 'call' or similar instruction, assuming that current Battle.snp variations do not use such patterns in their call to the module. This only slightly raises the bar for an attacker, though; he or she would only need pick a more specific location in Battle.snp through which to stage a call, such as the actual location used in normal calls to the Lockdown module.

3.6 Limited Pool of Challenge/Response Tuples

Presently, the Battle.net servers contain a fairly limited pool of possible challenge/response pairs for the version check and authentication system. Observations suggest that most products have a pool of around one thousand values that can be sent to clients. This has been used against Battle.net in the past, which was countered by an increase to 20000 possible values for several Battle.net products. Even with 20000 possible values, though, it is still possible to capture a large number of logon attempts over time and build a lookup table of possible values. This is an attractive option for an attacker, as he or she need only perform passive analysis over a period of time in order to construct a database capable of logging on to Battle.net with a fairly high success rate. Given the relative infrequency of updates to the pool of version check values (typically once per patch), this is considered to be a fairly viable method for an attacker to bypass the version check and authentication system.

This limitation could easily be addressed by Blizzard, however, such as through the implementation of one or more of the below suggestions:

1. Periodically rotate the set of possible version check values so as to ensure that a database of challenge/response pairs would quickly expire and need to be rebuilt. Combined with a large pool of possible values, this approach would greatly reduce the practicality of this attack. Unfortunately, the author suspects that this would require manual intervention each time the pools were to be rotated by the part of Blizzard in the current Battle.net server implementation.

2. Implement dynamic generation of pool values at runtime on each Battle.net server. This would

require the server to have access to the requisite client binaries, but is not expected to be a major challenge (especially since the author suspects that Battle.net is powered by Windows already, which would allow the existing Lockdown module code to be cleaned up and repackaged for use on the server as well). This could be implemented as a pool of possible values that is simply stirred every so often; new challenge/response values need not necessarily be generated on each logon attempt (and doing so would have undesirable performance implications in any case).

4 Conclusion

Although the Lockdown module and associated authentication system represent a major break in Blizzard's ongoing battle against non-genuine Battle.net client software, there are still many improvements that could be made in a future release of the version check and authentication system which would fit within the constraints imposed on the version check system, and still pose a significant challenge to an adversary attempting to spoof Battle.net logons using a non-genuine clients. The author would encourage Blizzard to consider and implement enhancements akin to those described in this paper, particularly protections that overlap and complement each other (such as the debug register clearing and memory checksum schemes).

In the vein of improving the Lockdown system, the author would like to stress the following principles as especially important in creating a system that is difficult to defeat and yet still workable and viable from a development and deployment perspective:

1. Defense in depth with respect to the various protection mechanisms in place within the module is a must. Protection systems need to be designed to complement and reinforce eachother, such that an attacker must defeat a number of layers of protection schemes for any one significant attack to succeed to the point of being a break in the system.

2. Countermeasures intended to frustrate reverse engineering or easy duplication of critical algorithms need to be viewed in the light of what an adversary might do in order to 'attack' (or duplicate, re-implement, or whatnot) a 'guarded' (or otherwise important) algorithm or section of code. For example, an attacker could ease the work of reimplementing parts of an algorithm or function of interest by wholesale copying of assembler code into a different module, or by loading an "authentic" module and making direct calls into internal functions (or the middle of internal functions) in an effort to bypass "upstream" protection checks. Keeping with this line of thinking, it would be advisible to interleave protection checks with code that performs actual useful work to a certain degree, such that it is less trivial for an adversary to bypass protection checks that are entirely done "up front" (leaving the remainder of a secret algorithm or function relatively "vulnerable", if the check code is skipped entirely).

3. Countermeasures intended to create "time sinks" for an adversary need to be carefully designed such that they are not easily bypassed. For instance, in the current Lockdown module implementation, there are twenty flavors of the Lockdown module; yet, in this implementation, it is trivially easy for an adversary to discover the differences (in a largely programmatic fashion), making this "time sink" highly ineffective, as the time for an adversary to breach it is likely much less than the time for the original developers to have created it.

4. Measures that depend on external, imported APIs are often relatively easy for an attacker to quickly pinpoint and disable (for example, the method that debug register breakpoints are disabled by the Lockdown module is immediately obvious to an adversary, if they are even the least bit familiar with the Win32 API (which must be assumed). In some cases (such as with the debug register breakpoint clearing code), this

cannot be avoided, but in others (such as validation of module base addresses), the same effect could be potentially implemented by use of less-obvious approaches (for example manually traversing the loaded module list by locating the PEB and the loader data structures from the backlink pointer in the current thread's TEB). The author would encourage the developers of additional defensive measures to reduce dependencies on easily-noticible external APIs as much as possible (balanced, of course, against the need for maintainable code that executes on all supported platforms). In some instances, such as the manual resolution of Storm symbols, the current system does do a fair job of avoiding easily-detectable external API use.

All things considered, the Lockdown system represents a major step forward in the vein of guarding Battle.net from unauthorized clients. Even so, there is still plenty of room for improvements in potential future revisions of the system. The author hopes that this article may prove useful in the strengthening of future defensive systems, by virtue of a thorough accounting of the strengths and weaknesses in the current Lockdown module (and pointed suggestions as to how to repair certain weaker mechanisms in the current implementation).