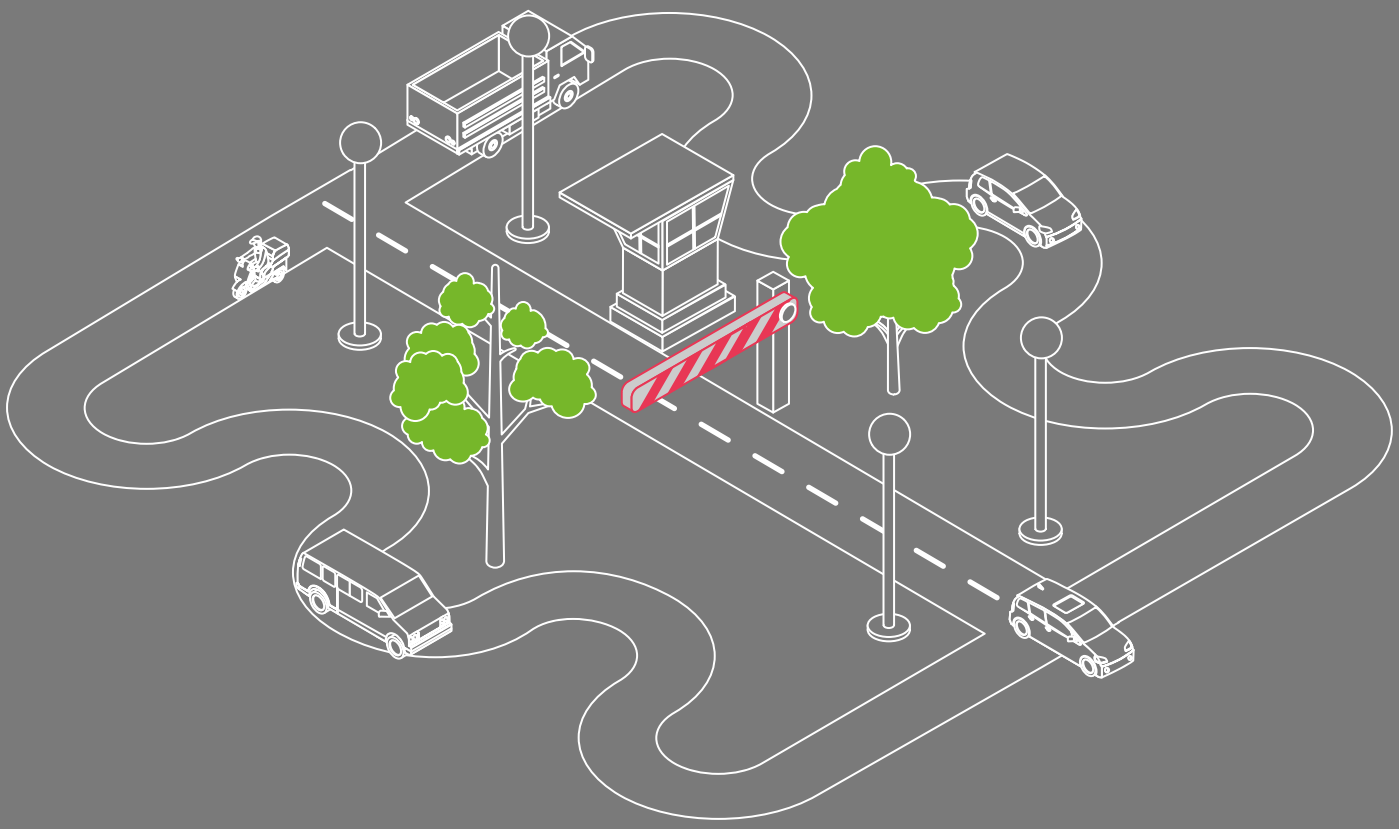
EMBEDI



Hack ATM with an anti-hacking feature and walk away with $1M in 2 minutes

Contents

Introduction

The Embedi team focuses not only on the security of embedded/smart devices and firmware for computers but also on critical devices, such as ATMs. ATMs consist of various devices with their own firmware. Application Control solutions fall into the type of software that appeals to our interests the most. These are now widely available on the market, but attacks on ATMs have so far been successfully pursued. We have become interested in how this software actually protects ATMs and does it make it harder for hackers to get to the cash.

ATMs

In general, the subject of our research is ATM security.

We will regard an ATM simply as a safe deposit, which is controlled by a computer. Currency is put into boxes, which are loaded into two devices in the safe: one for withdrawal (dispenser) and another for deposit (bunch note acceptor). The computer is connected to a card processing server through an isolated network.

There are many people involved in the making of, installing, and operating ATMs. Potentially, they can exploit their access for theft. These people and their capabilities are reflected in a typical model of threats to a bank owning ATMs.

- Internal violators:
  - Software developers: creating backdoors and errors in code
  - Contractors: handing cryptographic keys over to attackers
  - Service engineers: spoofing hardware and software components, malicious use of keys, negligence (leaving a safe open)
  - Cash-In-Transit guards: stealing currency boxes
- External violators:
  - Bank clients: manipulation of banknotes during cash-in (gluing, threads, etc.) and cashout (retrieving a part of a stack of notes)
  - Attackers without expertise: theft of an ATM, attack on cash-in-transit guards, social engineering
  - Attackers with expertise and mechanical tools: destruction of the device, accessing the safe, manipulation of the deposit slot
  - Attackers with expertise and hardware and software for local influence: skimming, Black Box attack, card cloning, accessing the PC inside the ATM
  - Attackers with expertise and hardware and software for remote influence: unauthorized access through the local network, malware installing, exploitation of software and OS vulnerabilities

Our expertise covers threats coming impacting programmable components of a device. In this article, we analyze the capabilities of an attack that exploits software vulnerabilities on the ATM's built-in PC. Exploitation of these vulnerabilities should lead to arbitrary code execution at the highest level of the execution environment.

The benefit of executing your own code on a built-in computer is that it allows sending commands to the dispenser, what usually happens each time we insert a card. However, our code, unlike built-in software, leaves out all irrelevant details, such as entering a PIN or requesting balance. It is just about cash – and all at once.

DEMO clip: https://www.youtube.com/watch?v=wIodOC1iP5Q

ISS specifics in ATMs

One of the key points of ensuring a secure operation of an ATM is to protect the integrity of the system. Information Security System (ISS) suppresses any attempts to modify the system or add external components to it: both software (executable files, software installation packages, scripts) and hardware (USB storage, CD/DVD devices). The goal is to prevent any extraneous code from running on the device. The system, in an unchanged way, must implement only its own functionality. That sounds reasonable, because, as practice shows, there are many ways to upload malware to the ATM PC:

- In local access:
  - To connect to the USB bus, by breaking out the camera, or to make holes at the location of the wires
  - To open the body of the ATM unnoticed and use a copied service key and insert a disc, a flash drive, a wireless keyboard plug
- In remote access:
  - To upload malware through a remote desktop from the bank
  - To exploit vulnerabilities in one of the many network services that are installed to monitor and manage the device

In addition, access control is equally important. Access to devices, processes, file system, registry objects, and other elements are controlled by the system, and with a correct configuration an attacker only has limited privileges at the initial stage of the attack. Having bypassed the integrity control described above, an attacker will still be dealing with the needto escalate privileges to access critical resources, such as the dispenser.

The market offers a number of products designed to mitigate risk for the banks which operate ATMs. Many of these are made by well-known AV-vendors and include technologies they have already developed: AV-scanner with signatures, heuristics, emulators, and unpackers; HIPS (proactive protection, behavior analysis); firewall. Given the specific security requirements for this class of devices, the products also have more severe countermeasures such as application whitelisting and device control – generally described above as integrity control tools – as well as secondary access control tools (as related to the OS native tools).

We have studied some of these ISS components and reached intriguing conclusions about their work. As an example of such product, let's take a look at Kaspersky Embedded Systems Security (KESS).

Overview of KESS features

In its minimal configuration, the solution performs system integrity control. There are two modules to accomplish this task:

- Device Control – controls connection of USB storage devices, such as flash drives, external HDDs, MTP Media Player devices, etc
- Applications Launch Control – controls code running in the system. There are some rules for:
  - Running processes and loading libraries
  - Running scripts and MSI packages. As a rule, objects are identified by digital signatures, hashes, and paths

KESS complies to Default Deny policy: everything not explicitly permitted is forbidden.

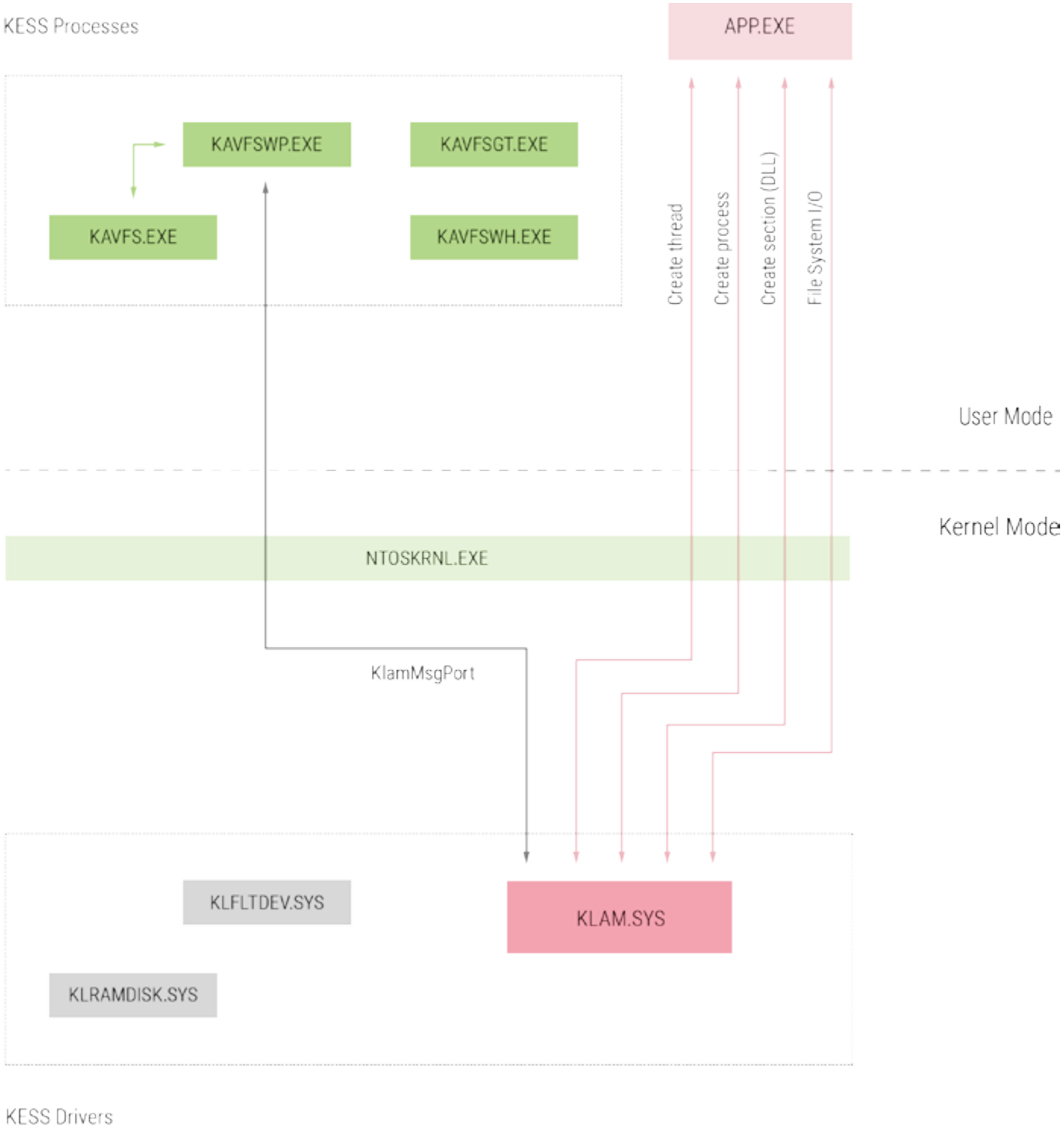
Optionally, you can add Real-Time File Protection – antivirus protection. According to its signature bases and heuristics, the AV scanner will check files when they are accessed.

For more information about the product, see the presentation: http://www.bts.md/BTS_files/Kaspersky%20Embedded%20Systems%20Security.pdf

KESS Architecture

The system is represented at two levels:

- Kernel Mode: In this privileged kernel execution mode, KESS drivers are operating
- User Mode: here KESS service processes are running among user and system processes



The components in the diagram include:

- The «Kaspersky Lab Interceptor and Activity Monitor» KLAM.SYS. driver. This registers a filter for the file system and gains control over callbacks when file system objects are accessed. Registers notify routines to create processes and threads. Thus, it performs system-wide monitoring and control at kernel mode. The driver is a part of Application Launch Control and Real-Time File Protection
- The «Kaspersky Lab PNP Device Filter» KLFLTDEV.SYS. driver. This controls the use of PNP devices marked as «Mass Storage»; a part of Device Control
- The «Kaspersky Lab Virtual RAM Disk» KLRAMDISK.SYS. driver
- KlamMsgPort message channel of the KLAM.SYS driver and the KAVFSWP.EXE. process. A variety of requests from the user mode are passed through this channel to the driver
- The KAVFS.EXE service and the KAVFSWP.EXE child processes. The logic of ISS checks and solutions is centered here. Verdicts are passed to the driver and actions of the monitored processes are allowed or blocked (for example, loading a DLL)
- The KAVFSWH.EXE service process: «Provides interconnection between the application and the external Agent to perform process memory protection»
- KAVFSGT.EXE service process: «Provides KESS management functionality»

Attack on KESS

We will try to bypass this security system and achieve privileged execution of our code on the machine, which should allow us to use the connected dispenser. According to the structure of the solutions offered in the product, our attack strategy has the following stages:

- First, having local access, we will clear the way to enter our data on the PC bypassing Device Control. We will bypass the filter that prevents third-party devices from connecting and put our binaries and scripts onto the host for the further work
- Then, we will bypass Applications Launch Control to run our code on the machine
- And the last step: We will escalate the privileges of our process in the Windows access control system by exploiting a vulnerability in KESS. This will give us unrestricted access to the system's objects

Bypassing KESS

Bypassing Device Control

KESS limits the use of USB drives. If this feature is enabled and correctly configured, it prevents attackers from using their tools.

However, it is the only class of devices that are subject to control.

If the ATM's main PC has a physically accessible CD drive, it will not prevent an attacker from uploading files outside. The same goes for access through a Remote Desktop.

If no attempts succeed, then there are other ways to bypass, including:

- USB, PS/2 keyboards, keyboard emulators – here will help:
  - A VBS script or built-in Certutil utility to decode printed characters into binary data
  - The old DEBUG debugger that can be used as a hex editor
- Free COM ports can also be used for data transfer, and are available from the cmd command line

Thus, obvious ways to enter data on the machine are left. Adding a keyboard to the whitelist is not likely to happen because it complicates maintenance service.

Bypassing Launch Control

The KLAM.SYS driver captures launch of executables, scripts, and library sections loads. After that, it sends them for a check (under the specified rules) to the KAVFSWP.Exe. If it is a component from outside, the use of it will not be allowed.

PowerShell

There are built-in interpreters in the OS, such as WScript, which can be used despite the prohibition of arbitrary script execution. For example, it can be done like this:

```
mshta.exe vbscript:close(eval(CreateObject("Scripting.FileSystemObject").OpenTextFile("1.vbs").ReadAll()))
```

The code from the file will be executed in the interpreter, and you can read and write files and registry keys, start processes, etc. (see Windows Script Host). It is interesting, but it will not be enough to work with the dispenser. A path to run the native code is needed.

PowerShell is much more flexible in this regard, and it has direct access to the system API. To run a shellcode, it is trivial to call VirtualAlloc, WriteProcessMemory, and CreateThread. You can solve the problem of script execution prohibition the same way:

```
powershell.exe Invoke-Expression $(Get-Content hello.ps1)
```

However, PowerShell is not usually installed on machines with Windows XP. Here you can use the following bypass.

Signed Executables

By default, a rule is created that allows running binaries with digital signatures trusted in the OS. This applies to all Windows components. For example, you can use a debugger signed by Microsoft to inject a shellcode into a legitimate process. The NTSD debugger goes with Windows XP; there is a signed debug library, dbgeng.dll, in all versions of Windows.

NTVDM

KESS does not create any obstacles to running DOS executables (EXE, COM). However, it was found that the NTVDM is a known way to escalate privileges.
http://web.archive.org/web/20160309015106/archives.neohapsis.com/archives/fulldisclosure/2010-01/0346.html

Timeout

Recently, an attack scenario for this component of KESS was published. It aimed to exhaust system resources by running a large number of binary instances in parallel, which do not have to pass the whitelist to be run. This attack would overload the verification module so that system calls to start processes were not processed within the timeout period, and after that the process was allowed to start without waiting for the KESS decision.

This way, you can achieve arbitrary code execution bypassing Device Control and Applications Launch Control whitelists in local or remote access to the PC. The next step is to escalate privileges to bypass the OS access control.

Operation of KESS: escalating privileges

Vulnerability

During the investigation of the KESS KLAM.SYS driver code, an interesting area was found:

```
void *__thiscall create_module_list(PEB_LDR_DATA *peb_ldr, unsigned int *out_buf_ptr,
unsigned int *out_buf_sz_ptr)
{
  void *result; // eax@1
  PVOID buf; // ebx@1
  char *module_list; // edi@2
  int i; // eax@3
  int module_list_entry; // esi@9
  size_t strlen; // ecx@11
  void *v9; // [sp+10h] [bp-30h]@1
  int buf_1; // [sp+1Ch] [bp-24h]@9
  unsigned int offs; // [sp+20h] [bp-20h]@9
  unsigned int sz; // [sp+24h] [bp-1Ch]@1
  CPPEH_RECORD ms_exc; // [sp+28h] [bp-18h]@2

  result = 0xC000000D;
  v9 = 0xC000000D;
  buf = 0;
  sz = 0;
  *out_buf_sz_ptr = 0;
  *out_buf_ptr = 0;
  if ( peb_ldr
```

```
 {
    ms_exc.registration.TryLevel = 0;
    module_list = &peb_ldr->InMemoryOrderModuleList;
    if ( *module_list != module_list )
    {
      for ( i = *module_list; i != module_list; i = *i )
        sz += 0x24 + *(i + 0x1C) + 0xA;
      if ( sz )
      {
        sz += 0x1008;
        buf = ExAllocatePoolWithTag(PagedPool, sz + 0x1000, 'imLK');
      }
      if ( buf )
      {
        buf_1 = buf;
        offs = 0;
        for ( module_list_entry = *module_list; module_list_entry != module_list;
module_list_entry = *module_list_entry )
        {
          offs += 12;
          offs += 12;
          offs += 12;
          strlen = *(module_list_entry + 28);
          offs += strlen + 10;
          if ( offs > sz )
            break;
          memcpy_special_0(strlen, 3, &buf_1, *(module_list_entry + 0x20));// see at src
buf!
          memcpy_special(4u, 7, &buf_1, (module_list_entry + 16));
          memcpy_special(4u, 8, &buf_1, (module_list_entry + 24));
          memcpy_special(4u, 9, &buf_1, (module_list_entry + 20));
        }
        offs += 8;
        memcpy_special(0, 0, &buf_1, 0);
        sz = offs;
      }
    }
    ms_exc.registration.TryLevel = -2;
    if ( buf )
    {
      *out_buf_ptr = buf;
      *out_buf_sz_ptr = sz;
      v9 = 0;
    }
    result = v9;
  }
  return result;
}
```

Next, ExAllocatePoolWithTag is called, and the size of the allocation sz + 0x1000 is requested
The list is being passed for the second time – from beginning to end – and, for each module, a structure containing data from the list is added to the allocated buffer. The offset in the buffer is then compared to the sz value (and not to sz + 0x1000 - see (3)) to protect it from overflowing.
At the end, 8 more zeros are added to the buffer; its pointer and size are returned.

This calculation of the buffer size has a possibility of integer overflow. What we can do is:

1. to form such a InMemoryOrderModuleList_fake linked list that the lengths of strings and the remaining coefficients in the sum calculated in the first cycle equal 0xffffdff8;
2. place it in the PEB;
3. trigger list review by this function;
4. at step (2) of operation algorithm, the function will be assigned a value sz + = 0x1008 (Sz = 0xFFFFDFF + 0x1008 = = 0xFFFFF000);
5. to call the ExAllocatePoolWithTag, the size parameter of Sz_arg = sz + 0x1000 (sz_arg = 0xFFFFF000 + 0x1000 = = 0) – a zero-size buffer will be allocated;
6. the data from our InMemoryOrderModuleList_fake will then be copied into this buffer until the offset value offs > sz (sz == 0xFFFFF000), that is, far more than the allocated zero of bytes:)

We are free to build a InMemoryOrderModuleList_fake for any value of sz. In the range of 0xFFFFDFF8 <= sz < 0xFFFFEFF8, we can allocate a buffer the size of 0 to 0x1000 bytes and overflow it.

Overflow control

To use the overflow, we need to solve the next problems:

1. The data will be copied to the buffer until the offset value offs> sz, where sz is close to the maximum unsigned integer, i.e., about 4 GB - this should be stopped.
2. Make the driver execute this function on our exploit.
3. Position the object controlled by us into the kernel pool after the overflowing buffer to avoid damage to random kernel objects.

Controlling the overflow size

Copying 4 GB of data is DoS.

The first idea was to replace the InMemoryOrderModuleList pointer while executing the vulnerable function so that the value of sz would be taken to overflow it; and in the second run, of copying, the list would already be of a more appropriate size to fill the allocated memory.
This is possible to achieve, taking advantage of the fact that the function implements double-

fetch in relation to our data. When the driver passes our InMemoryOrderModuleList lists, it does not stop the process, and we can write a different pointer to the PEB. It is not possible to seize the moment appropriate for the replacement, so we can just run the cycle, change one value to another and hope for luck. It works, but it is very unstable.

Another convenient way was found by chance. We have noticed that the system has not crashed into the blue screen once when it was copying data string into the buffer from the invalid pointer to the module path. The point is that it cannot be seen in a decompiled listing of the function, but there is a handler for the exception:

```
PAGE:B1879590 loc_B1879590:                           ; DATA XREF:
.rdata:off_B184CDD0no
PAGE:B1879590                 Mov     esp,    [ebp+ms_exc.old_esp] ; Exception handler
0 for function B17E8452
PAGE:B1879593                 Mov     ebx,    [ebp+P]
PAGE:B1879596                 Test    ebx,    ebx
PAGE:B1879598                 Jz      short   loc_B18795AB
PAGE:B187959A                 Push    0       ; Tag
PAGE:B187959C                 Push    ebx     ; P
PAGE:B187959D                 Call    ds:ExFreePoolWithTag
PAGE:B18795A3                 Xor     ebx, ebx
PAGE:B18795A5                 Mov     [ebp+P], ebx
PAGE:B18795A8                 And     [ebp+sz], ebx
PAGE:B18795AB
```

When the code makes the memcpy-rep movs, and the ESI register contains a pointer to the memory that can't be read, control is handed over here; the buffer is being freed; a return from the function is made, and everything continues. We can prepare the data to be copied to the buffer so that the module path string ends when we need to stop on the page with the PAGE_NOACCESS attribute. This way, we control length and content of the overflow accurately and reliably.

Trigger for function

It was suggested that this function could be called by the KESS service in the event of discovery of a certain malware, that could inject its libraries into some processes. However, when scanning a large array of different samples, no such triggers were found.

It is convenient to perform a vulnerable function by running a memory scan:

```
kavshell.exe scan /memory
```

However, a user without administrator rights cannot do it. We have reversed kavshell.exe and, allegedly, the rights check for running a scan is located in the KESS service and not in its interface, which leaves us no opportunity to make a request for such a scan bypassing rights check. The function is called for each process at the start of KESS, but you also need rights to stop it. You can make the service restart by sending a sample, which will cause a DoS when unpacked, to the scanner's engine.

The time to use the scan is when the system begins. The exploit can be put into the autorun area which is available to the current user, and then it will restart the machine. As soon as the exploit is run, the scan will work and impact our process too. However, it will take more time to make a spray in the pool to control the order of allocations; and, as the experiments have shown, we will not be able to make it during this time window. We will do the following: all processes are being scanned – from the newest back to the oldest one. The exploit launches 500 calculators in suspended mode, and we are buying the time that the KESS kernel needs to work with these processes.

Paged pool spray

To place the object that we are controlling following the overflowing buffer, we will try to create the desired kernel memory state. We need to understand that the vulnerable buffer is being allocated in a paged pool. This is a little inconvenient for the following reasons:

- The objects that we can use to quickly gain control when overwriting callbacks are allocated in a nonpaged pool - see. A convenient example;
- There are few paged pools, and they are switched by the allocator to balance the load.

To understand further details, it is highly recommended that you familiarize yourself with the Windows kernel allocator architecture.

Filling

By connecting the debugger and inspecting the PoolDescriptor.ListHeads lists of paged pools, one notices that at the time of the KESS memory scan, after Windows boots, there are allocation sizes that have not yet been involved in intensive system initialization processes. For example, by allocating blocks of 0x400 bytes, we can assume that the same blocks will not be allocated and freed during our work, creating errors in the spray and compromising the exploit's reliability. You can allocate such blocks in paged pools by creating named objects, such as events.The object name string consists of WCHAR characters and is then placed into the same pool as the onebeing used by the KESS driver to create module lists. We can set the size of the vulnerable buffer and the size of the strings with event names, to get them into same PoolDescriptor.ListHeads lists.

We also need to allocate an array of blocks in several paged pools. Experiments show that creating objects one-by-one in the exploit cycle causes the same pool to be reused to store objects' name strings, whereas, to call the allocator in a vulnerable driver function, a block may return unpredicted in any one of the several pools in the system. We add a small delay after each 1000th object is created in our exploit and this time is usually enough to switch the pool index in the allocator. All pools are being filled.

Next, we make single-block holes in a spray, destroying some of the objects that have been created earlier. Free blocks of the chosen size are returned to the PoolDescriptor.ListHeads lists and then they are waiting to be allocated by the KESS driver.

Use of overflow

Because this part of the exploit is architecture specific, we will use Windows 7x32 as our target system.

There are different techniques of exploiting overflows in a pool. We will overwrite the Poolindex in the next block. When a block is freed, this value is adopted as an index of PoolDescriptor, to the corresponding ListHead of which the freed block will be added.

PoolDescriptor contains PendingFrees list, that is, blocks waiting to be added to ListHeads. First, ExFreePoolWithTag calls ExDeferredFreePool inside itself to free PendingFrees: merge free neighboring blocks and attach them to a ListHead of the desired size. Then, control is returned from ExDeferredFreePool to ExFreePoolWithTag and the block freed by this function is added to PendingFrees.

When PoolIndex is rewritten with a value higher than there are pools created by the system (in our case, 5), a call for ExDeferredFreePool will adopt NULL value – uninitialized address from the array, where addresses for PoolDescriptor are added when pools are created. ExDeferredFreePool will have to dereference NULL and work according to its algorithm with PendingFrees and other members of the structure at this invalid address.

For Windows 7, the NULL pointer dereference is an option that must be used. Using NtAllocateVirtualMemory system API, we can select pages, starting with NULL, making this area of memory suitable for reading and writing. On these pages, we carefully build a fake PoolDescriptor with appropriate values for all members so that ExDeferredFreePool works without errors.

ExDeferredFreePool will take our fake PoolDesciptor and pass the PendingFrees list. It will take a block from the list, check its and the adjacent blocks' headers, and insert the block's address into a corresponding ListHead of our pool. Here lies the key objective of the entire operation. To add an item to a linked list, the code in ExDeferredFreePool will take a pointer stored in our descriptor and write a value from our descriptor according to the pointer – the freed block's address from the Pendingfrees.

So, we have led the rewriting of the header of the allocation next to our buffer to write an arbitrary value to an arbitrary address.

Use of writing to an arbitrary address

With this base, we can record the shellcode address in some callback and achieve kernel-mode execution. Here a classic token stealing shellcode could navigate through the list of processes in the system, take the address to access the token and fit it into our process, escalating the access level to the maximum «NT AUTHORITY/SYSTEM.»

However, we will employ a different method. There is NtQuerySystemInformationsystem API, which will write for us the information on all handles in the system for the SystemHandleInformation parameter. For each handle, the address of the kernel object to which it refers is disclosed. We can get the token handle of our process using:

```
OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &htoken)
```

This object is interesting as a target for rewriting because it contains fields that define our access rights in the system.

By setting a single bit in the token whose address we are going to find, we can give our process the SeDebugPrivilege privilege without the kernel code control flow hijacking.

Use of SeDebugPrivilege

Our process, within the system-wide debugging rights, can read, write, and execute code in other processes, including the system ones. Through WriteProcessMemory and CreateRemoteThread, it is easy to make a trivial injection into any process outside our limited user processes.

The subtleties remain:

- Starting with Windows 7, we cannot create a new thread in a process outside of our session using CreateRemoteThread. The desired aim would be the KESS service process:) Within the session, winlogon will do, and our code will get in it the same «NT AUTHORITY/SYSTEM» token.
- One would think, what could be more alarming for HIPS than an injection through WriteProcessMemory and CreateRemoteThread? However, KESS does not engage in such behavior analysis and totally tolerates it.

Thus, you can escalate privileges exploiting a vulnerability in KESS.

Timeline of Disclosure

Exploit demo

https://www.youtube.com/watch?v=-yzCJwUuofE

Full vector for an attacker

Taking into account the weaknesses identified and the vulnerability found in the product, we believe that such a plan is appropriate for an attack:

1. A hole in the plastic panel of the ATM body is made; behind it, the USB bus is accessed.It would be ideal if you could get to the computer itself.

2. A keyboard emulator is connected. Notepad is opened; a zip archive encoded in base64 is typed in with tools for our further work. The encoded archive is saved.

3. A VBS script is typed to decode the archive into a binary form, the script is run, and the archive is unpacked.

4. The exploit is recorded in the registry for autorun. The computer is rebooted.

5. You have to choose the best option so that you do not have to carry any extras in the archive. 5.1. If the target machine is running Windows XP, the NTSD debugger runs with the script we brought. Using the debugger managed by the script, we inject the shellcode into some process, e.g., Calc. 5.2. If the target machine is running a higher OS version, then PowerShell is run with another script, which runs shellcode in its memory using VirtualAlloc, WriteProcessMemory, and CreateThread.

6. Shellcode reads into memory and runs the main part of the exploit to escalate privileges exploiting a vulnerability in the KESS driver.

7. The exploit worked, and the attacker gets into the system with “NT AUTHORITY/SYSTEM” rights. This is where the last piece of code works. We can send commands to the dispenser in two ways:

   7.1. Form packets by ourselves and send them to the driver for sending to the device.

   7.2. Using the universal and documented XFS interface, which hides hardware-dependent moments of communication with devices. This is done in the majority of the known ATM malware: Tyupkin, Atmitch, GreenDispenser, Suceful.

Conclusion

Our analysis shows that the software, which does not consist of a built-in protection or a protection implied in the OS design, is easy to bypass because of the incompatibility of the properties of the system in which it is executed with and the security properties that are implied in the software. The OS is not designed to maintain that boundary.

An attempt to statically filter executables into trusted and untrusted (or malicious) files does not preclude the risks of manipulating the state of the machine.

In addition, the complex logic of classifying files performed by such a privileged entity extends attack surface, and therefore introduces additional vulnerabilities to the system with common defects in its code.

Contacts

Telephone: +1 5103232636
Email: info@embedi.com

Address: 2001 Addison Street
Berkeley, California 94704

www.facebook.com/Embedi
twitter.com/_embedi_

linkedin.com/company/embedi
website: embedi.com