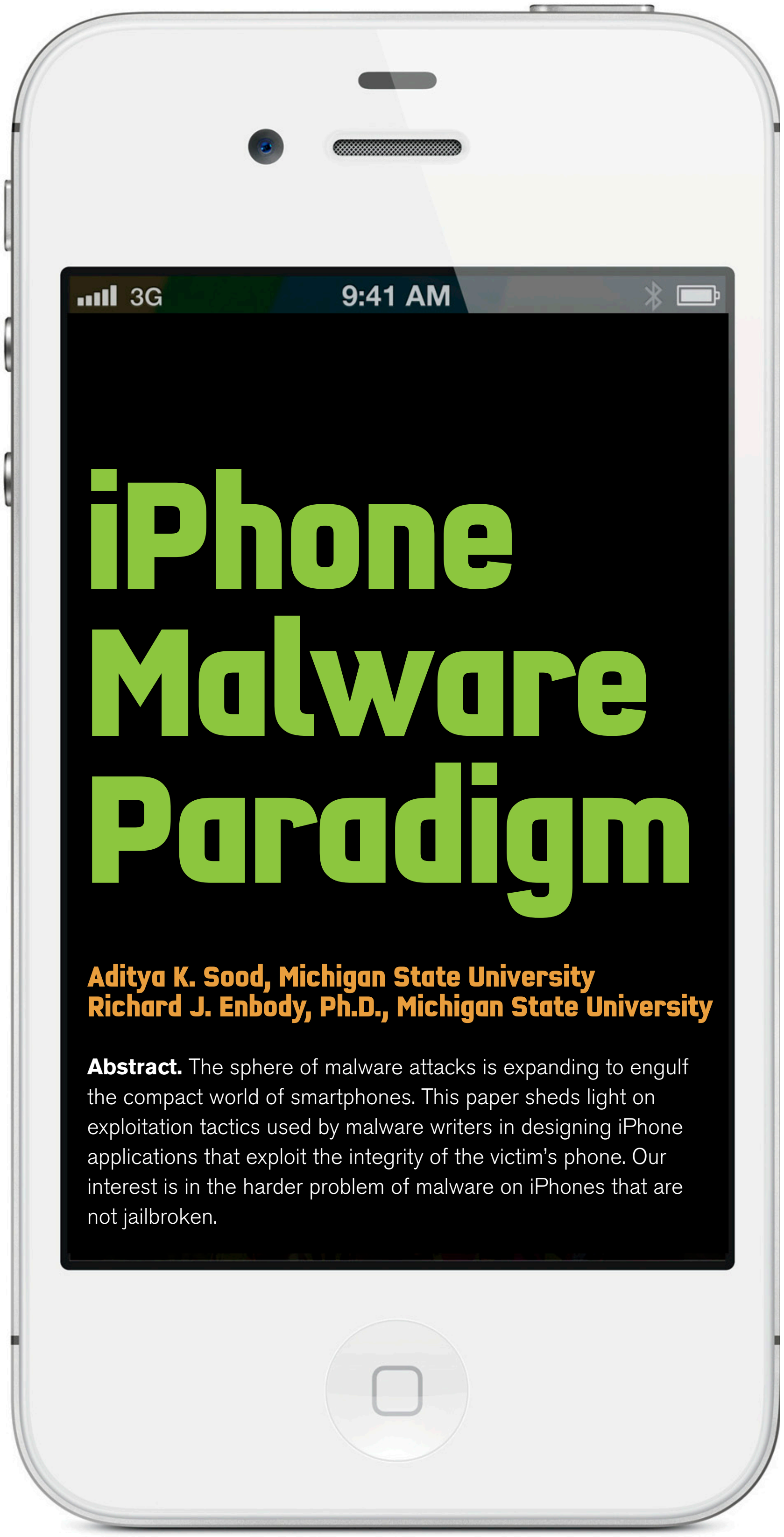


iPhone Malware Paradigm

Aditya K. Sood, Michigan State University
Richard J. Enbody, Ph.D., Michigan State University

Abstract. The sphere of malware attacks is expanding to engulf the compact world of smartphones. This paper sheds light on exploitation tactics used by malware writers in designing iPhone applications that exploit the integrity of the victim's phone. Our interest is in the harder problem of malware on iPhones that are not jailbroken.

Introduction

Malware has begun infecting the mobile world. Several studies [1, 2] have been conducted showing how mobile malware is exploiting the online world. Android malware infections are exploding as compared to iPhone. The primary reason is that Android is an open source platform where as iPhone's iOS is closed. Our target is to discuss the potential possibilities of malware occurrence in iPhone devices. In spite of the iPhone's strong security platform, malware is making inroads. However, successful iPhone exploitation depends on several factors. As we know, Apple has implemented several security barricades in order to secure the iPhone environment aided by tight control of their app market. Apple considers iPhones marginalized by the jailbreaking process as unsecure since all the inherent protection mechanisms have been circumvented by the attacker.

Is it possible to write a malicious application that may not exploit security vulnerability, but can still perform some spyware activity? The answer is yes. This is possible in certain scenarios where a malicious application can be designed to bypass Apple's application review process to execute illegitimate operations on an user's iPhone. In this paper, we discuss practical scenarios and effective techniques that can be used to host malicious applications on non-jailbroken Apple iPhones.

Understanding Apple's iPhone Applied Security Model

Apple enforces strict security features in order to protect the integrity of iOS. Its security model has the following features:

• With the advent of iOS 4, Apple introduced a new data protection procedure in which stored data is secured using hardware encryption. The device stores the user passcode key on an internal chip using 256-bit encryption. The Unique ID (UID) of the devices is used as a key to encrypt a file on iPhone.

• The iOS environment is divided into two main partitions. Similar to UNIX, the root partition manages the kernel and base OS. The user partition contains third-party applications and data. All applications run in a user mode with a standard set of access rights and built-in restrictions. The iOS system-level binaries are related to OS X and Darwin. In order to preserve the integrity of applications, Apple implements a code signing process [8]. The code signature consists of three parts. First, the signature consists of a UID that is present in the info.plist files under CFBundleIdentifier structure. Second, it requires a seal that is built from hashes and checksums of various files and other components of the application bundle. Third, it requires a digital signature. All the signatures are stored in the MACH-O header format. Code signing code verification is implemented in a kernel level using the execv () command.

• Third-party applications running on iOS are sandboxed [9]. This concept is implemented to force privilege separation among different components in iOS. It means that third-party applications are not able to run code at kernel level—a secure practice to avoid exploitation of privileges. The application sandbox is implemented using three techniques. First, entitlements which decide the functionality of the application. Second, containers that provide an application directory for supplying read/write operations. Third, powerbox which provides a secure way to open and handle dialog boxes. Together these three methods collaboratively form the application sandbox. Of greatest interest to malware writers, third-party applications are not allowed to interact with kernel-level extensions.

Anatomy of Jailbreaking

For completeness, let us take a brief look at jailbreaking. This attack exploits vulnerabilities in browser, plugin, and iOS components to take control of a victim's iOS device. As a result, jailbreaking [3, 4] culminates in a complete compromise of the iOS device. It primarily uses security vulnerabilities that provide root control of the device. Once the vulnerability is exploited, the attacker is able to run his native code and turn the victim's iOS device into a weapon. Jailbreaking also deploys code signing bypass mechanisms [5] in order to install open source packages such as Cydia [6]. It is also possible to spread malware after jailbreaking. In 2009, a default SSH password vulnerability was exploited on jailbroken iPhones to propagate the iKee [7] worm and its variants.

iPhone Malware–Exploitation Model

A malware infection in an iPhone can be categorized into three distinct classes:

• The first class of malware results from exploitation of security vulnerabilities to get root-level access. Jailbreaking falls into this category. Once rooted, attackers can start services on the iPhone to turn into a malicious entity for spreading malware. In this case, the attacker has to target a specific set of victims. It is difficult because it becomes an action by choice whether the user wants to jailbreak his or her iPhone or not. As a result, attackers force the user to visit a malicious domain using social media tricks to download the malicious code. In a real-time environment, it is hard to spread this class of malware on a large scale as there is a trust layer that Apple provides its users by having applications hosted on Apple's online store. The malware exploits the root privileges as the kernel is already compromised after the exploitation of the security vulnerability. iPhone rootkits [10] are also classified into this class. For example, the Dutch iPhone ransomware [11] belongs to this category of malware.

• The second class of malware exploits the default security model of Apple. This is basically exploited by spyware applications that look legitimate and bypass Apple's App Store verification process. Once in the App Store, infection is easier as the malicious application can be easily disseminated to a number of iOS users. The malicious application might not be able to compromise the kernel as it runs in the sandbox, but it can definitely steal users' sensitive information, history, address book contacts, and so on. This class of malware is a classic example of iPhone spyware that exploits the trust boundary between the user and App Store. For example, SpyPhone [12, 13] falls into this category of malware.

• The third type of malware is a hybrid of both classes of malware discussed above. Hybrid malware is triggered through a generic application that is hosted on the App Store. When a user downloads it, at first it looks legitimate but behind the scenes it starts sending texts to the phone numbers listed in the contacts directory of the victim's iPhone. The text itself carries a link to a malicious website that serves a jailbreaking code. Drive-by download attacks are used extensively for spreading this class of malware. For example, iSAM [14] is a hybrid class of iPhone malware.

The lifecycle of mobile malware is presented in Figure 1.

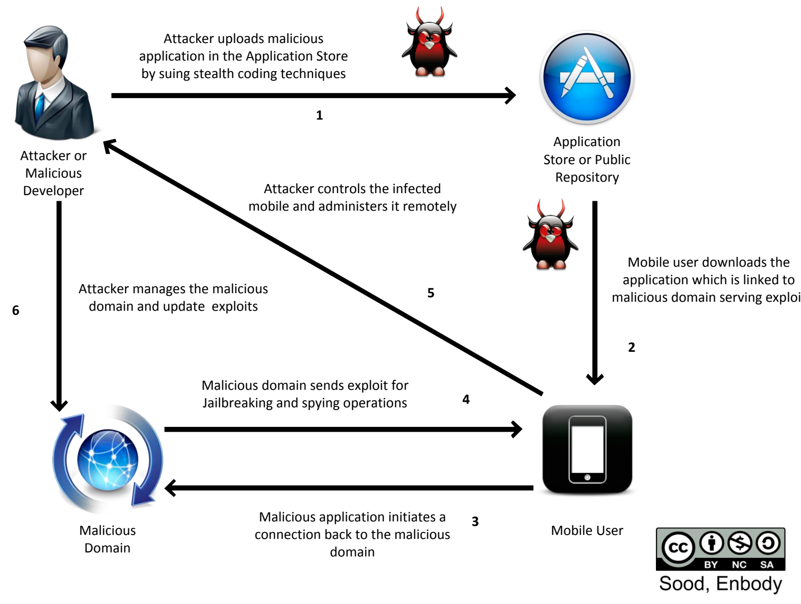
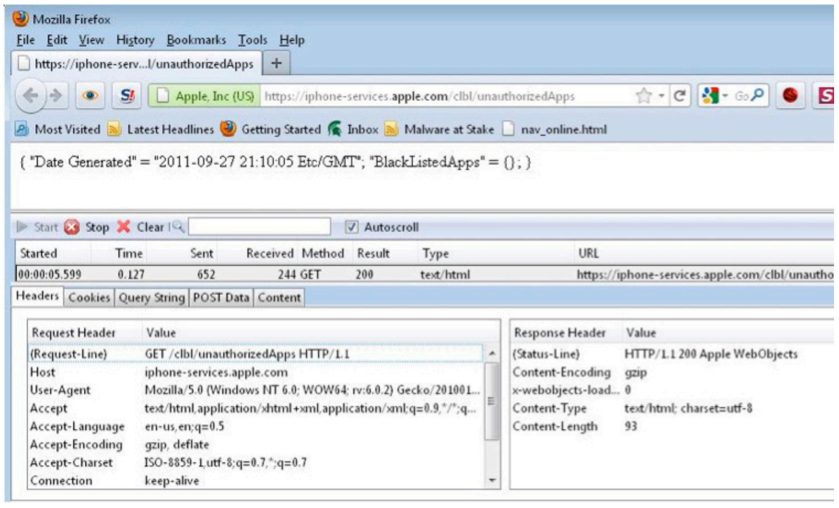

Attacker uploads malicious application in the Application Store by suing stealth coding techniques
1
Attacker or Malicious Developer
Application Store or Public Repository
Attacker controls the infected mobile and administers it remotely
Mobile user downloads the application which is linked to malicious domain serving exploit
Attacker manages the malicious domain and update exploits
6
5
2
Malicious domain sends exploit for Jailbreaking and spying operations
4
Malicious application initiates a connection back to the malicious domain
3
Malicious Domain
Mobile User
BY NC SA
Sood, Enbody

Figure 1: Lifecycle of Mobile Malware

Inside the Apple Kill Switch–Remotely Deactivating Applications

iOS has the built-in protection of a kill switch [15, 16] that enables Apple to kill a malicious application that does not comply with its policies. Applications installed on the iPhone regularly correspond back to the App Store to provide updates about the state of the device. Apple uses blacklisting with a list of applications that are malicious and should be turned off remotely. It is kept in the "unauthorzeapps" file on an Apple server. We performed a quick check on a required URL in order to see which applications are blacklisted. Figure 2 shows that currently there are no applications marked as unauthorized.



```
{ "Date Generated" = "2011-09-27 21:10:05 Etc/GMT"; "BlackListedApps" = (); }
```

Started	Time	Sent	Received	Method	Result	Type	URL
00:00:05.599	0.127	652	244	GET	200	text/html	https://iphone-services.apple.com/clbl/unautho

Request Header	Value
(Request-Line)	GET /clbl/unauthorizedApps HTTP/1.1
Host	iphone-services.apple.com
User-Agent	Mozilla/5.0 (Windows NT 6.0; WOW64; rv:6.0.2) Gecko/201001...
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q...
Accept-Language	en-us,en;q=0.5
Accept-Encoding	gzip, deflate
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection	keep-alive

Response Header	Value
(Status-Line)	HTTP/1.1 200 Apple WebObjects
Content-Encoding	gzip
x-webobjects-load...	0
Content-Type	text/html; charset=utf-8
Content-Length	93

Figure 2: Blacklisted–Unauthorized Apps Check

This functionality is distinct from removing the applications from the App Store because this procedure is designed to deactivate rogue applications remotely. It seems like Apple usually removes the application directly from the App Store. However, the remote deactivation process exists as a proactive defense.

Listing 1: Sandbox profiles

```
kSBXProfileNoNetwork (= "nonet")
kSBXProfileNoInternet (= "nointernet")
kSBXProfilePureComputation (= "pure-computation")
kSBXProfileNoWriteExceptTemporary (= "write-tmp-only")
kSBXProfileNoWrite (= "nowrite")
```

Listing 2: Obfuscation using NSString Object

```
(NSString *)obfuscate_code:(NSString *)string withKey:(NSString *)
key {
 // Create data object from the string
NSData *data = [string dataUsingEncoding:NSUTF8StringEncoding];
char *code_ptr = (char *) [raw_data bytes];

// Mapping the pointer to key data
char *k_data = (char *) [[key dataUsingEncoding:NSUTF8StringEncoding] bytes];
char *key_ptr = k_data;
int key_index = 0;

// For each character in data, xor with current value in key
for (int x = 0; x < [raw_data length]; x++) {

// Apply XOR operation on every character
 *code_ptr = *code_ptr++ ^ *key_ptr++;
 if (++key_index == [key_length]) key_index = 0, key_ptr = k_data; }
  return [[[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding] autorelease];}
```

App Store Application Review–Dependencies and Reality

There are not many details available about Apple's app review procedures. However, based on a developer's view some details can be deduced. Some of the procedures implemented by the App Store are as follows:

• The App Store strictly requires a developer to be enrolled in the Apple's iPhone Developer program [19]. In order to get the approval, the developer has to submit a binary and not the source code, which in turn means that detailed source code analysis is not a part of the verification process. The App Store usually checks for user interface inconsistencies, private API calls and malware. However, malware scrutiny depends on the malware exploitation model mentioned earlier. It is hard to infer details of the Apple application review process, but dynamic and static analysis (pattern matching) are thought to be a part of the process. Given what we know about the review process, it is possible that stealthy programming techniques may be able to circumvent the detection modules.

• The Apple iPhone Licensing Agreement [20] requires a developer not to perform reverse engineering tactics on the applications hosted on the App Store and software developer kit components. Based on this fact, it seems reasonable to assume that Apple itself is following this practice and is not performing reverse engineering on submitted applications. In practice, it is not feasible to reverse engineer the thousands of applications submitted on a weekly basis.

• Most mobile malware aims to steal a user's data at the application layer. In spite of Apple's restrictive policies, default access to user data is available to any application running on an iPhone. The sandboxed environment prevents applications from interacting with each other but a malicious application can subvert the trust boundary of another application. In addition, the sandbox facilitates the process of preventing the background activities that are possible in jailbroken iPhones. Listing 1 shows the different set of sandbox profiles [22] available.

• Malicious applications have been hosted in the App Store in the past. For example, Aurora Feint [17] was considered malicious because the application uploaded users' contacts to the developer's server which is a straightforward breach of privacy. Another example is Pinch Media [18] that followed the same practices of breaching privacy.

Obfuscation–Bypassing Blacklisting

Obfuscation can be useful for legitimate developers as well as for malware writers. Obfuscation is used to prevent the exposure of API functionality. For example, best practices suggest avoiding the embedding of hard-coded credentials in the application. However, developers sometimes hide keys in the code using obfuscation or store credentials on a webserver and rewrite queries after verification. That is, developers implement obfuscation modules for security purposes. Such code needs to pass security testing. Apple requires the application to be robust in nature. As long as the iPhone application is stable and does not crash, the App Store easily accepts an application having obfuscated modules.

While obfuscation is used by legitimate developers to prevent information leakage, a malware writer can use obfuscation to bypass the App Store verification process.

Most of the static analysis tools use blacklisting in which a certain set of strings are blacklisted. When the scanner runs the application code it matches blacklist patterns using regular expressions. Knowing this, it is possible to bypass the static analysis tool using obfuscation. Let us consider an example; In iPhone applications, strings are declared as NSString [21] which are immutable and represented as an array of Unicode characters. Listing 2 shows a prototype of implementing obfuscation using NSString object.

It is possible to obfuscate the strings in an iPhone application and then deobfuscate them at run time. There are many algorithms to perform this functionality. However, the XOR operation is an effective way of obfuscating strings. Generally, the following steps can result in implementation of obfuscated code in iPhone applications:

• The first step is to create a data object from the required string.

• The second step involves the declaration of pointers to the data and encryption key to be obfuscated.

• The third step involves the implementation of counter that runs through every character in a string and embeds a key using the XOR operation.

Code Hiding in Objective-C and Symbols Stripping

Apple is very strict in its review policy about using private API functions that are not documented because these hidden methods can be used by malware. Generally, applications using private API functions are rejected by the App Store. Objective-C does not provide support for private methods, but it is still possible to write methods that hide malicious code. Below are the two most widely implemented steps:

• Objective-C has a dynamic resolution feature in which a method is bound during compile time. The attacker can define a secret function whose signature matches Objective-C implementation. The secret function is declared in the class method. When that method gets called for the first time, the malicious code is bound to the class privately. This type of procedure is used to circumvent code detection using a tool such as Class-Dump. Listing 3 shows a code prototype that uses dynamic method resolution.

However, a skilled analyst may be able to figure out the presence of stealth code. For example, running Otool on a particular method results in the list of selectors that are used by the respective method. However, it is possible to obfuscate the method by generating selectors at run time using "*NSSelectorFromString()*" functions.

• In Objective-C, it is also possible to create functions that work similarly to instance methods. It means functions can access instance variables easily. These types of functions should be defined in the class implementation. It is not a normal way of doing things, but the desired method never appears in the Objective-C run time which hampers verification. Listing 4 shows the declaration of malicious function *hide_me* with instance variables. The function *hide_me* does not have its own selector rather it uses the selector of *stealth* instance (public) method defined in the class.

The two methods discussed provide a way to design code which can hide from tools that examine code so they can be accepted by the App Store.

Additionally, stripping is a technique used in UNIX platforms to remove unnecessary information from a binary and object files to improve performance. A malicious developer can use stripping to remove information prior to submission of an application binary to the App Store. Doing so removes clues that might indicate the malicious nature of the code.

Exploiting the Remote Server End Points

Generally, all iPhone applications communicate back with a webserver (HTTP End Point) in order to exchange data between the application and the server on a regular basis. It is possible for malware to exploit the HTTP end point mechanism. At the time of verification, Apple performs a behavioral analysis of the application and scrutinizes the communication pattern. At the time of submission, the attacker can make the HTTP end point legitimate and once approved by Apple, the same HTTP end point can be used to serve the exploit code which is downloaded into the victim's phone when the application interacts with a remote server. For example: consider the following scenario:

• Attacker writes an application that interacts with a remote server on the URL <http://www.mal-app-test.com/error.asp >. The error.asp webpage validates the resource and if that resource is not present then it raises an error.

• During the verification process, Apple finds it legitimate and the application is treated as good enough to host on the App Store.

• Once the application is hosted, it is possible to manipulate the "error.asp" webpage to deliver exploit code that is downloaded into the device and performs malicious functions.

This is a legitimate scenario that can be exploited to trigger malware infections in an iPhone.

Listing 3: Code hiding using dynamic resolution

```
// Setting a Class Interface as  Secret
@interface Secret ()
// secret function is defined in the class method
void hide_me(id self, SEL _command);
@end

// Implementing Class
@implementation Secret
@synthesize handle;

// Selecting hide_me secret function and binding into the class method
+ (BOOL)resolveInstanceMethod: (SEL)aSel {
   if (aSel == @selector(hide_me)) {
      class_addMethod(self, aSel, (IMP)hide_me, "v@:");
      return YES;
   }
   return [super resolveInstanceMethod: aSel];
}

// This is an Instance Method holding a reference to hide_me
- (void)stealth {[self hide_me];}
   void hide_me(id self, SEL _command) {
   HIDE(@"Inside hide_me: %d", (LMethod *)self)->handle);
}
@end

//Class Dump Output
@interface Secret : NSObject { int handle; }

// Tool does not provide information about hide_me after static discovery of Class Method
+ (BOOL)resolveInstanceMethod:(SEL)arg1;
@property(nonatomic) int handle; // @synthesize handle;

// Class Dump only lists the Instance Method
- (void)stealth;
@end
```

Listing 4: Code hiding function variables as instance methods

```
(void)stealth { hide_me(self, _cmd);}
```

Cautionary Steps

Users play a critical role in the success of malware. There are a number of steps that a user can follow to reduce risk. These proactive steps are applicable to every smartphone whether Android or iPhone and are discussed as follows:

• Mobile users should not install any unauthorized application from third-party resources. The installed applications must be verified and authorized from legitimate vendors.

• The users should think twice prior to clicking any URL from non-legitimate resources. For example: users should be careful while chatting on social media applications such as Facebook and Twitter. Push notification messages should be scrutinized critically prior to executing any action based on the information in a message. E-mail attachments should not be opened directly until the user is sure about legitimacy.

• It is always advised to install anti-virus software on your mobile device which scans the device for potential suspicious activities and notifying users about changes in the system.

• Usage of strong passwords and avoidance of default security policies is always preferred.

* Users should carefully analyze the behavior of their mobile phones against any types of anomalous activities such battery drainage, high Internet data usage, and slower execution of applications.

Conclusion

In this paper, we have discussed the state of iPhone malware. There is no doubt that Apple has designed a robust verification policy but it is still possible to create stealthy malware that can bypass Apple's verification process. However, doing so requires devising a malicious application in an intelligent way using stealthy techniques such as code obfuscation, stripping, and code hiding. We believe that malware poses an increasingly serious challenge to the security of our devices and we need to be proactive in our defenses to ensure the security of our data and privacy. ✦

ABOUT THE AUTHORS

Aditya K. Sood is a senior security researcher and Ph.D. candidate at Michigan State University. He has worked in the security domain for Armorize, COSEINC and KPMG. He is also a founder of SecNiche Security Labs, an independent security research arena for cutting edge computer security research. At SecNiche, he also acts as an independent researcher and security practitioner for providing services including software security and malware analysis. He has been an active speaker at industry conferences and already spoken at RSA, Virus Bulletin, HackInTheBox, ToorCon, HackerHalted, Source, TRISC, AAVAR, EuSecwest, XCON, Troopers, OWASP AppSec USA, FOSS, CERT-IN, etc. He has written content for HITB Ezine, Hakin9, ISSA, ISACA, CrossTalk, Usenix Login, and Elsevier Journals such as NESE and CFS. He is also a co-author for Debugged magazine.

E-mail: adi.zerok@gmail.com
E-mail: soodadit@cse.msu.edu
Phone: 517-755-9911

Richard J. Enbody, Ph.D., is associate professor in the Department of Computer Science and Engineering at Michigan State University where he joined the faculty in 1987. He has served as acting and associate chair of the department and as director of the computer engineering undergraduate program. His research interests include computer security; computer architecture; web-based distance education; and parallel processing, especially the application of parallel processing to computational science problems. Enbody has two patents pending on hardware buffer-overflow protection that will prevent most computer worms and viruses.

Email: enbody@cse.msu.edu
Phone: 517-353-3389

REFERENCES

1. Malware Goes Mobile, http://www.cs.virginia.edu/~robins/Malware_Goes_Mobile.pdf
2. Mobile Malware Madness and How to Cap the Mad Hatters, https://media.blackhat.com/bh-us-11/Daswani/BH_US_11_Daswani_Mobile_Malware_Slides.pdf
3. IPhone Jailbreak: The Ultimate Guide, http://www.appleiphonereview.com/iphone-jailbreak/iphone-jailbreak/
4. IPhone Hacks Jailbreak, http://www.iphonehacks.com/jailbreak_iphone
5. Bypassing iPhone Code Signatures, http://www.saurik.com/id/8
6. How to Use Cydia: A Walkthrough, http://appadvice.com/appnn/2008/07/how-to-use-cydia-a-walkthrough
7. An Analysis of the iKee.B (Duh) iPhone Botnet, http://mtc.sri.com/iPhone/
8. Code Signing, http://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html
9. Application Sandbox, http://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/ApplicationSandboxing/ApplicationSandboxing.html#//apple_ref/doc/uid/TP40005929-CH6-SW2
10. iPhone Rootkits, http://www.ekoparty.org/archive/2010/ekoparty_2010-Monti-iphone_rootkit.pdf
11. Hacker Holds Dutch iPhones for Petty Ransom, http://www.wired.com/gadgetlab/2009/11/iphone-hacker/
12. SpyPhone iPhone App Can Harvest Personal Data, http://threatpost.com/en_us/blogs/spyphone-iphone-app-can-harvest-personal-data-120409
13. IPhone Privacy, http://seriot.ch/resources/talks_papers/iPhonePrivacy.pdf
14. iSAM: An iPhone Stealth Airborne Malware, http://www.icsd.aegean.gr/publication_files/conference/462488002.pdf
15. Apple iPhone 'kill switch' discovered, http://www.telegraph.co.uk/technology/3358115/Apple-iPhone-kill-switch-discovered.html
16. Apple's Jobs confirms iPhone 'kill switch', http://www.telegraph.co.uk/technology/3358134/Apples-Jobs-confirms-iPhone-kill-switch.html
17. Aurora Feint iPhone App Delisted For Lousy Security Practices, http://gizmodo.com/5028459/aurora-feint-iphone-app-delisted-for-lousy-security-practices
18. iPhone App Store Secrets - Pinch Media, http://www.slideshare.net/pinchmedia/iphone-appstore-secrets-pinch-media
19. Apple Developer Program, http://developer.apple.com/programs/start/standard/
20. iPhone Developer License Agreement, https://www.eff.org/files/20100127_iphone_dev_agr.pdf
21. NSString Class Reference, http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/Reference/NSString.html
22. Sandbox, http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man3/sandbox_init.3.html

Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications, is seeking dynamic individuals to fill several positions in the areas of software assurance, information technology, network engineering, telecommunications, electrical engineering, program management and analysis, budget and finance, research and development, and public affairs.

To learn more about the DHS Office of Cybersecurity and Communications and to find out how to apply for a vacant position, please go to USAJOBS at www.usajobs.gov or visit us at www.DHS.GOV; follow the link Find Career Opportunities, and then select Cybersecurity under Featured Mission Areas.