Predictive Markets: A Software Security Application

Jon Oakley, Kushal Hebbar, Nathan Tusing, Carl Worley
Holcombe Department of Electrical and Computer Engineering
Clemson University
Clemson, USA
{joakley,khebbar,ntusing,cworley}g.clemson.edu

***Abstract*—Accountability is the biggest challenge facing software security. We propose a software security approach based on Augur's predictive markets. We anticipate this will introduce a financial disincentive for developers to produce secure code. By leveraging Augur's existing framework, we can automate the creation of software markets where users will decide whether or not they believe a specific software version is secure. We extend Augur's framework by providing 1) a toolkit for the automatic creation of software security markets, 2) trading bots that simulate market activity, 3) modifications to Augur's UI that incorporate real-time information, and 4) a streamlined development environment. Our goal is to provide consumers with a metric that lets them easily evaluate software security, as well as provide feedback for the developers.**

***Index Terms*—Predictive Markets, Augur, Software Security**

I. Introduction

Accountability is the biggest challenge facing software security. Financial accountability ensures there are fiscal repercussions for negligent security vulnerabilities. One of the most important factors affecting software security is how well the software is written. Well written software is clean, efficient, and uses techniques that reduce the overall complexity of the code–reducing the attack surface. Unfortunately, most software companies lack appropriate incentives to produce well-written code. If a company produces software with security vulnerabilities, they release patches before the software's vulnerability are publicly revealed (but not necessarily before the vulnerabilities are exploited). Companies often make enough money to offset the cost of the vulnerability, which actually provides a financial disincentive towards developing secure software.

This work presents the following contributions in the pursuit of secure software:

1) The idea of using predictive markets for software security.
2) A toolkit for the automatic creation of software security markets.
3) Trading bots to simulate market activity for demonstration purposes.
4) Modifications to the existing user interface (UI) that includes real-time information.
5) A development environment that allows new developers to quickly and efficiently begin contributing.

Our goal is to leverage Augur's predictive market framework to provide consumers with a metric for software security. This metric will allow consumers to make informed decisions and provide software developers with impactful feedback.

II. Background

Security is a vital aspect of all aspects of technology, whether it be application security, software security, or network security. Technology today has advanced immensely. With every emerging technology introduced, our life gets easier. However, along with these technological advancements come attacks such as identity thefts, frauds, Man-in-the-middle (MitM) attacks, and Denial-of-service (DoS) attacks. These technological advancements are highly dependent on software–every business relies on the internet and computer network for operations. Therefore, software security must be given top priority.

Software security is the concept of building software so it functions and produces accurate results under all conditions. As software developers are becoming increasingly aware of software vulnerabilities, they are adapting and evolving a set of best practices to address attacks on security. Some of the best practices include building abuse cases, collecting security requirements from clients, performing risk analysis, external review, and carrying out static analysis [1].

Another aspect of software security that assists in protecting a system against attacks are software security standards such as OWASP (Open Web Application Security Project), which provides a structure for standards and best practices that are a vital aspect of software development [2]. OWASP has developed the SAMM (Software Assurance Maturity Model) [3] to assist such security measures. NIST (National Institute of Standards and Technology) [4] is another security standard that provides a common language and definition structure for software development. Another such standard is the CWE (Common Weakness Enumeration) which provides a common language to define threats [5]. There is a constant increase in security threats as the rate of software developments increase. These software security standards act as a pillar to develop secure and productive applications.

A. Predictive Markets

Predictive markets are based on the idea of swarm intelligence, which is a concept that taps into the collective minds of species to make decisions, answer questions, and predict the future. This field has been studied in flocks of birds, schools of fishes, and swarms of bees, and hives of ants. However, unlike these animals, humans lack the ability to form connections through subtle actions and pheromones. Unanimous AI is an implementation of Swarm AI that allows user to connect through their web browsers and work collectively and in parallel [6]. Unanimous AI allows users to to weigh options and converge on a collective decision in real-time.

An experimental analysis with a group of users who claim to be sports enthusiasts were asked to predict the outcome of 200 National Hockey League games over 20 weeks [7]. In addition to predicting the outcome of each game, users were asked to predict if the outcome of self-assessments of the teams were accurate, these teams were termed the "pick of the week". The Artificial Swarm Intelligence (ASI) system predicted the "pick of the week" team with 85% accuracy, as opposed to 62% accuracy of the Vegas odds [7].

Predictive markets on the other hand runs on the concept of wisdom of crowds, such as voting, polls and surveys. Fundamentally, wisdom of crowds works on individuals providing their input in isolation in comparison to individuals working collectively to converge on solutions. Prior research concludes that human swarms results in better outcomes with much higher accuracy than traditional wisdom of crowds. Current swarm AI technology is modeled based on a honeybee swarm (hive mind), as they can effectively make critical decisions.

Prediction markets are financial markets where the payout depends on the outcome of a future event. Sports betting is a simple example of a prediction market. The odds reflect the number of participants who have bet on a particular outcome, and all the participant who bet on the winning outcome get a *share* of money staked by all participants proportional to the size of their initial stake. The market closes when the event begins, as this marks the start of the outcome. The relative prices of the various shares in relation to the payout of those shares is called the *odds*, and these odds represent the probability of each outcome. Since the efficient market hypothesis states that share prices in a market will reflect all available information [8], a prediction market will reflect the sum knowledge of its participants. In contrast to traditional solutions based on the wisdom of crowds, these markets provide real-time feedback to users in the form of price fluctuations, just like the feedback given in [6]. Empirical studies have shown prediction markets to be at least as accurate as a panel of experts–for instance, prediction markets have often outperformed national polls regarding presidential elections [9].

III. Architecture

The predictive market security framework leverages three main pieces of technology: Ethereum, Augur, a frontend marketplace. Augur's trading platform is built on Ethereum's smart contract framework, and Augur's APIs can be integrated with a marketplace websites to provide users with information about current trends and updated information. Figure 1 depicts the architecture, along with the novel contributions that extend Augur's functionality.

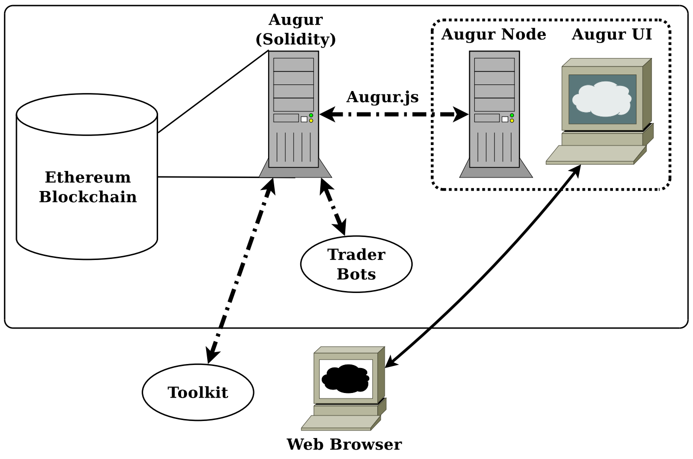


Fig. 1: The architecture of the software security security platform.

A. Ethereum

Ethereum is a cryptocurrency that extends traditional blockchain technology with smart contract applications [10]. Smart contracts are programs that are run on the blockchain in a special environment called an Ethereum Virtual Machine (EVM), which ensure that compatibility across all Ethereum nodes. Smart contracts guarantee the output of code through the same consensus that secures the blockchain. Ethereum uses Solidity as the programming language for it's smart contracts. There are serveral different implementations of Ethereum, and the Augur team chose `go-ethereum` (Geth). The Geth node had full functionality when interacting with the existing Ethereum blockchain, and it also provides a testnet (fully functional development blockchain). It is trivial to generate currency on the testnet, which makes it perfect for developing blockchain-based applications.

B. Augur and Smart Contract Templates

Augur [11] builds on this secure execution environment by creating libraries of Ethereum smart contracts that support token trading. Augur allows users to create *markets* for an *event*. When Alice creates a market on Augur, she puts Ether into two escrow accounts that will be returned if certain conditions are met. Alice also specifies the market duration and the types of tokens that will be traded, such as: event will happen, and event won't happen. When Bob believes the event will happen, he buys the initial *event will happen* tokens from the contract (the market created by Alice). Similarly, when other users buy the initial *event won't happen* tokens, they also buy them from the contract. This creates a fund held by the contract that will be paid out to the users whose tokens reflect reality. If the event doesn't happen, the fund is paid to all users who have *event will not happen* tokens.

Augur also supports trading tokens. So, Bob may not truly believe the event will happen, but he may think that others will believe the event will happen, and therefore, he may buy tokens and try to sell them for a higher price.

The final issue Augur addresses is handling the payout after a market closes. Any market that is defined so that the terms of resolution are not perfectly unambiguous can, in the reporting phase, be marked “INVALID”. In this case, no outcome is marked as “true” and the value of the shares is distributed equally among all shareholders. Since a controversial reporting of outcome can cause the Augur platform to fork into competing universes, and since the worst case scenario involves REP tokens (a token that indicates the reliability of a reporter) being migrated into multiple child universes, so that each universe has positive value less than the value of the parent universe, the REP holders are strongly incentivized to avoid forks at all costs. This likely means that INVALID outcomes occur extremely easily, so as to avoid possible controversy. Any market that may possible be INVALID will be ignored by traders, who are as likely to lose money as gain it.

There are several components to the Augur framework: Augur Core, Augur Node, and Augur.js. Augur Core is the smart contract framework deployed in Solidity that supports the token trading. These contracts are submitted to the Ethereum blockchain as transactions, which allow them to be run in the EVM. Augur.js is a Node.js API that provides access to the status of the Augur Core EVMs. Augur Nodes uses Augur.js to aggregate the complete state of Augur. The node acts as an intermediary between the UI and Augur Core.

C. Augur Marketplace

A frontend marketplace will provide users with easy access to all available software markets. In the future, user and developer profiles will also be linked to software to prevent parasitic markets (such as developers betting against their own software). The frontend will also link the markets with all available source code, documentation, and resources relevant to the market. News listings will also be available to provide users with the latest information. As an example of this, we include the commit stream for the git repository in the trading platform as an external source of information for traders.

IV. Experimental Setup

The development environment was a CentOS 7 (v7.5.1804) VM with 4GB of RAM. Python 2.7 was installed for utility scripts. Git (v1.8.3.1) was installed to clone repositories and help with automatic market creation. Docker (v1.13.1) was installed to run the Geth blockchain. `npm` (v5.6.0) and Yarn (v0.33.11) were installed to manage the Node.js packages. `nvm` (v0.33.11) was installed to manage the Node.js version (v9.11.1).

Augur.js (v5.10.3) was cloned, which included Augur Core (v1.1.0) and a Geth docker container (v1.8.10). The Augur Node repository (v6.0.3) was cloned, along with the Augur UI repository (v5.10.5). The script in Listing 1 was run to set up the Augur.js API and deploy the Geth blockchain.

```bash
#!/bin/bash
source ~/.bashrc
git clone https://github.com/AugurProject/augur.js.git
nvm use 9.11.1
cd augur.js
sed -i 's/^\s*createMarkets/\/\/createMarkets/' scripts/dp/index.js
npm install
npm run build
yarn link
npm run docker:pull
mkdir geth-data
sudo chcon -Rt svirt_sandbox_file_t $(pwd)/geth-data
docker run -e GETH_VERBOSITY=4 -it -p 8545:8545 -p 8546:8546 -v $(pwd)/geth-data:/geth/chain augurproject/dev-node-geth:latest
```

Listing 1: Bash script to build the Node.js API and set up the blockchain.

The script sources the user’s bashrc file which contains configuration information for `nvm`. The latest git repository for Augur.js is cloned, and the Node.js version 9.11.1 is specified. Next, the `dp` script is modified to prevent markets from automatically being deployed with Augur Core. Dependencies are installed with `npm`, the API is built, and yarn is used to create a link that other repositories can use.

Finally, the latest Geth docker container is pulled. A new data directory for Geth is created and allowed in SELinux, and the docker container is started. By creating a physical data directory for the Geth container, that blockchain is made persistent. The script in Listing 2 was used to deploy Augur Core and set up the Augur Node.

```bash
#!/bin/bash
source ~/.bashrc
git clone https://github.com/AugurProject/augur-node.git
nvm use 9.11.1
cd augur-node
npm install
yarn link augur.js
timeout 60 npx dp deploy
npm run clean-start
```

Listing 2: Bash script to build the Augur Node and deploy Augur Core.

After `nvm` is enabled and the repository is cloned, we specify Node.js version 9.11.1 for uniformity, and install the missing dependencies. Then, we use `yarn` to link the Augur Node project to the Augur.js API. This allows us to use functions and scripts in the Augur.js API. The `dp` script is then used to deploy Augur Core to the Geth blockchain, and the Augur Node server is started. The script in Listing 3 was used to start the Augur UI server.

```bash
#!/bin/bash
source ~/.bashrc
git clone https://github.com/AugurProject/augur-ui.git
nvm use 9.11.1
cd augur-ui
npm install
yarn link augur.js
yarn dev
```

Listing 3: The bash script to build and run the Augur UI server.

After `nvm` is enabled and the repository is cloned, we specify Node.js version 9.11.1 for uniformity, and install the missing dependencies. Then, we use `yarn` to link the Augur Node project to the Augur.js API and start the UI server. The scripts in Listing 4, Listing 5, and Listing 6 were used to restart the Geth blockchain, Augur Node, and Augur UI server, respectively

```bash
#!/bin/bash
cd augur.js
docker run -e GETH_VERBOSITY=4 -it -p
    8545:8545 -p 8546:8546 -v $(pwd)/
    geth-data:/geth/chain augurproject/
    dev-node-geth:latest
```

Listing 4: Bash script to restart the Geth blockchain.

```bash
#!/bin/bash
source ~/.bashrc
nvm use 9.11.1
cd augur-node
npm run clean-start
```

Listing 5: Bash script to restart the Augur Node.

```bash
#!/bin/bash
source ~/.bashrc
nvm use 9.11.1
cd augur-ui
yarn dev
```

Listing 6: Bash script to restart the Augur UI server.

Figure 2a shows the running blockchain. We can see blocks being mined along with other debug information. Figure 2b shows Augur Node running. We can see new blocks being processed along with JSONRPC queries to the Geth node.

V. Results

After successfully deploying the Augur framework, the contributions described in Section I were implemented. The market creation scripts described in Section V-A were designed and tested by Jon Oakley, the trading bots described in Section V-B were designed and tested by Carl Worley, the modifications to Augur's UI described in Section V-C were implemented and tested by Kushal Hebbar and Nathing Tusing, and the development environment described in Section V-D was developed and tested by Jon Oakley and Kushal Hebbar.

A. Market Creation

A Python wrapper `init-market.py` was developed for a Node.js program that use the Augur.js API to create a market for the specified git repository, as shown below.

```
$ init −market.py <git repo>
```

The Python wrapper prompted users for a category and tags for the market. The category is the security question being asked about the particular version of software, and the tag is a way to group different markets. Preliminary research has yielded the following market categories, along with their specific questions. These markets are time-limited, usually to two weeks.

1) Privilege Escalation: Does the referenced source code contain any vulnerabilities that knowingly (or unknowingly) leak the user's data to an untrusted third-party?
2) Privacy: Does the referenced source code contain any vulnerabilities that allow a malicious user to escalate their privileges (giving them access to any commands they would not otherwise have access to)?
3) Memory: Does the referenced source code contain any states where memory can be overwritten with content chosen by a malicious user?
4) Input: Does the reference source code contain any input sources that can be used to put the program in an unexpected state?
5) Race Conditions: Does the program ever try to access the same data from two different threads at the same time?

Figure 3 shows the process of creating a software security market from the command line. This version contains an abbreviated list of the market questions for demonstration purposes.

Figure 4 shows newly created market displayed on Augur's UI.

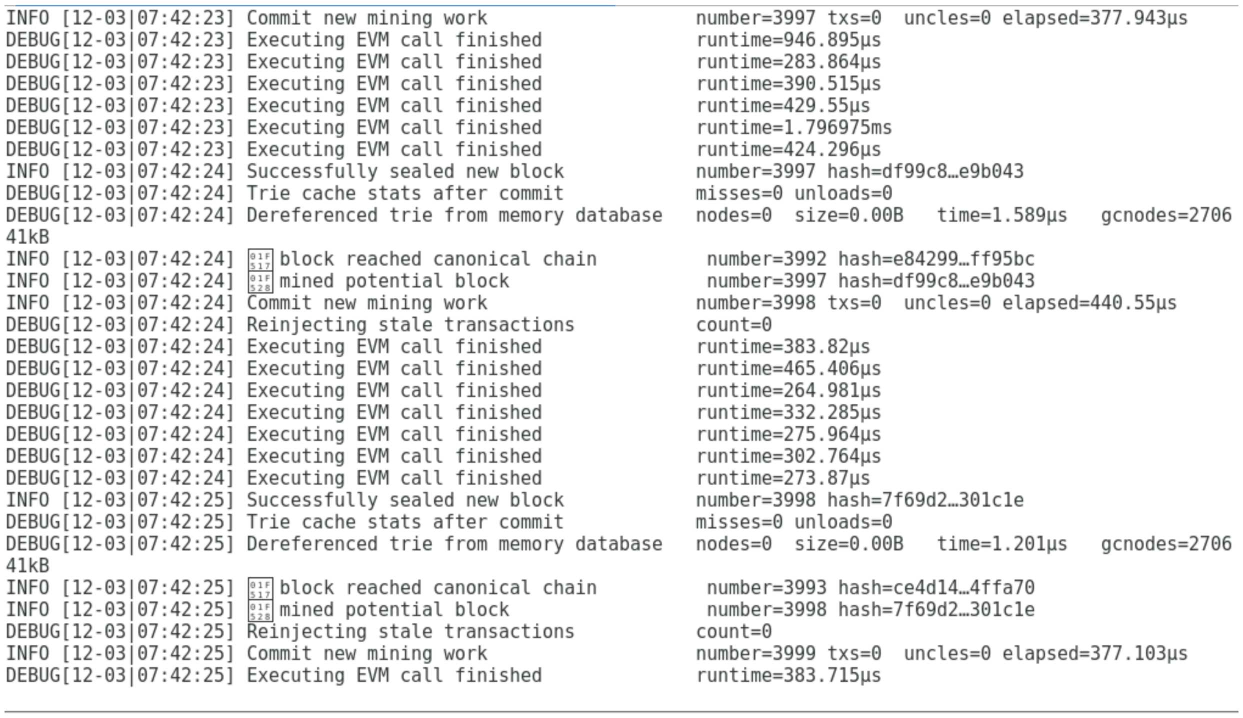
B. Trading Bots

The trading bots were developed to simulate real users on the market. This was done purely for demonstration purposes, as the security described in Section II-A is only present when a collection of knowledgeable users make trades based on informed decisions. The trading bots' functionality is shown in Algorithm 1.

Algorithm 1: The algorithm used simulate market trading data.

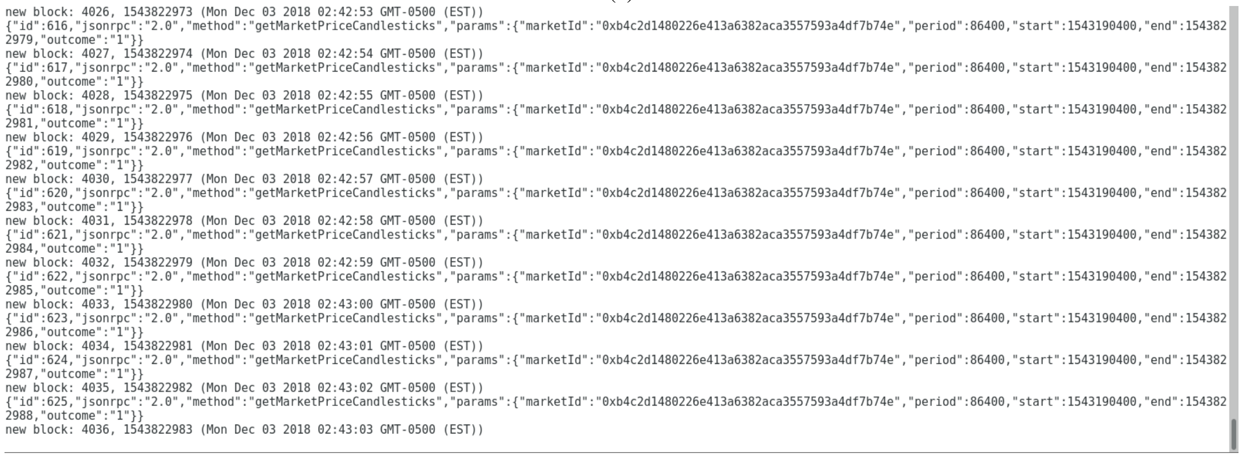
```
Name: Trading Bots
1 no_bug = True;
2 while no_bug do
3     tradeAtPrice(0.3);
4 end
5 Event BugFound(&no_bug)
6     no_bug = False;
7     tradeAtPrice(0.7);
```

Figure 5a shows the market operating under normal conditions, and Figure 5b shows the market after a potential bug has been discovered. In Figure 5a, the price is equivalent to the expected probability of a privacy vulnerability being found (36.56%) in go-ethereum (v1.8.19-4). Figure 5b shows the new probability of a privacy vulnerability (76.37%) after a simulated event has occurred.



```
INFO [12-03|07:42:23] Commit new mining work                   number=3997 txs=0  uncles=0 elapsed=377.943µs
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=946.895µs
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=283.864µs
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=390.515µs
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=429.55µs
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=1.796975ms
DEBUG[12-03|07:42:23] Executing EVM call finished              runtime=424.296µs
INFO [12-03|07:42:24] Successfully sealed new block            number=3997 hash=df99c8…e9b043
DEBUG[12-03|07:42:24] Trie cache stats after commit            misses=0 unloads=0
DEBUG[12-03|07:42:24] Dereferenced trie from memory database   nodes=0  size=0.00B   time=1.589µs   gcnodes=2706
41kB
INFO [12-03|07:42:24] 🔗 block reached canonical chain          number=3992 hash=e84299…ff95bc
INFO [12-03|07:42:24] 🔨 mined potential block                  number=3997 hash=df99c8…e9b043
INFO [12-03|07:42:24] Commit new mining work                   number=3998 txs=0  uncles=0 elapsed=440.55µs
DEBUG[12-03|07:42:24] Reinjecting stale transactions           count=0
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=383.82µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=465.406µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=264.981µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=332.285µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=275.964µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=302.764µs
DEBUG[12-03|07:42:24] Executing EVM call finished              runtime=273.87µs
INFO [12-03|07:42:25] Successfully sealed new block            number=3998 hash=7f69d2…301c1e
DEBUG[12-03|07:42:25] Trie cache stats after commit            misses=0 unloads=0
DEBUG[12-03|07:42:25] Dereferenced trie from memory database   nodes=0  size=0.00B   time=1.201µs   gcnodes=2706
41kB
INFO [12-03|07:42:25] 🔗 block reached canonical chain          number=3993 hash=ce4d14…4ffa70
INFO [12-03|07:42:25] 🔨 mined potential block                  number=3998 hash=7f69d2…301c1e
DEBUG[12-03|07:42:25] Reinjecting stale transactions           count=0
INFO [12-03|07:42:25] Commit new mining work                   number=3999 txs=0  uncles=0 elapsed=377.103µs
DEBUG[12-03|07:42:25] Executing EVM call finished              runtime=383.715µs
```

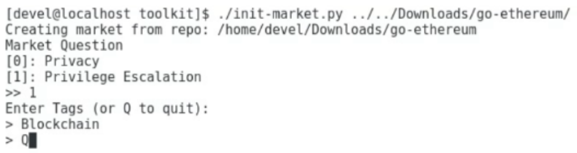
(a)



```
new block: 4026, 1543822973 (Mon Dec 03 2018 02:42:53 GMT-0500 (EST))
{"id":616,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2979,"outcome":"1"}}
new block: 4027, 1543822974 (Mon Dec 03 2018 02:42:54 GMT-0500 (EST))
{"id":617,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2980,"outcome":"1"}}
new block: 4028, 1543822975 (Mon Dec 03 2018 02:42:55 GMT-0500 (EST))
{"id":618,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2981,"outcome":"1"}}
new block: 4029, 1543822976 (Mon Dec 03 2018 02:42:56 GMT-0500 (EST))
{"id":619,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2982,"outcome":"1"}}
new block: 4030, 1543822977 (Mon Dec 03 2018 02:42:57 GMT-0500 (EST))
{"id":620,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2983,"outcome":"1"}}
new block: 4031, 1543822978 (Mon Dec 03 2018 02:42:58 GMT-0500 (EST))
{"id":621,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2984,"outcome":"1"}}
new block: 4032, 1543822979 (Mon Dec 03 2018 02:42:59 GMT-0500 (EST))
{"id":622,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2985,"outcome":"1"}}
new block: 4033, 1543822980 (Mon Dec 03 2018 02:43:00 GMT-0500 (EST))
{"id":623,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2986,"outcome":"1"}}
new block: 4034, 1543822981 (Mon Dec 03 2018 02:43:01 GMT-0500 (EST))
{"id":624,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2987,"outcome":"1"}}
new block: 4035, 1543822982 (Mon Dec 03 2018 02:43:02 GMT-0500 (EST))
{"id":625,"jsonrpc":"2.0","method":"getMarketPriceCandlesticks","params":{"marketId":"0xb4c2d1480226e413a6382aca3557593a4df7b74e","period":86400,"start":1543190400,"end":154382
2988,"outcome":"1"}}
new block: 4036, 1543822983 (Mon Dec 03 2018 02:43:03 GMT-0500 (EST))
```

(b)

Fig. 2: (2a) The Geth blockchain running in a docker container. (2b) The Augur Node aggregating results from the Augur Core EVM running on the blockchain.



```
[devel@localhost toolkit]$ ./init-market.py ../../Downloads/go-ethereum/
Creating market from repo: /home/devel/Downloads/go-ethereum
Market Question
[0]: Privacy
[1]: Privilege Escalation
>> 1
Enter Tags (or Q to quit):
> Blockchain
> Q
```

Fig. 3: An example market created from the command line using the Augur.js API.

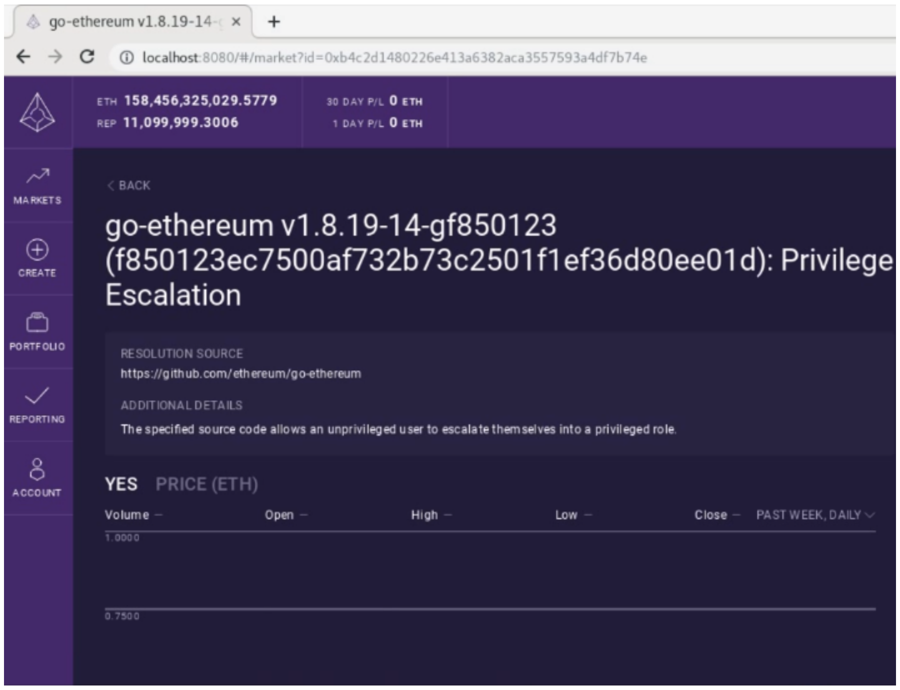
C. User Interface

A feed of recent commits is streamed from the github link in the market question. Figure 6 shows an example of this commit stream. The commits are updated every time the page is loaded. While not meant to be a comprehensive source of information, these commits provide useful information to traders regarding the overall health of a market.

D. Development Environment

As mentioned in Section IV, a VM was used as the primary development environment. This ensured all developers were working with the same system and prevented tedious compatibility issues. This VM is available for evaluation[1]. The

[1]https://www.dropbox.com/sh/11a2o18o0shi88h/AADpDpU5m_SAjjyMYX7fca3Ka?dl=0



go-ethereum v1.8.19-14-gf850123 (f850123ec7500af732b73c2501f1ef36d80ee01d): Privilege Escalation

RESOLUTION SOURCE
https://github.com/ethereum/go-ethereum

ADDITIONAL DETAILS
The specified source code allows an unprivileged user to escalate themselves into a privileged role.

Fig. 4: The newly created market displayed on Augur's UI after running the command line utility.

VM contains all of the source code discussed, and contains instructions for rapid deployment and testing. A large portion of time on this project was devoted to setting up a working Augur testnet, and this VM removes the need for other developers to make that substantial time investment. Giving developers more streamlined access should rapidly expand the number of domains where this style of security can be applied effectively.

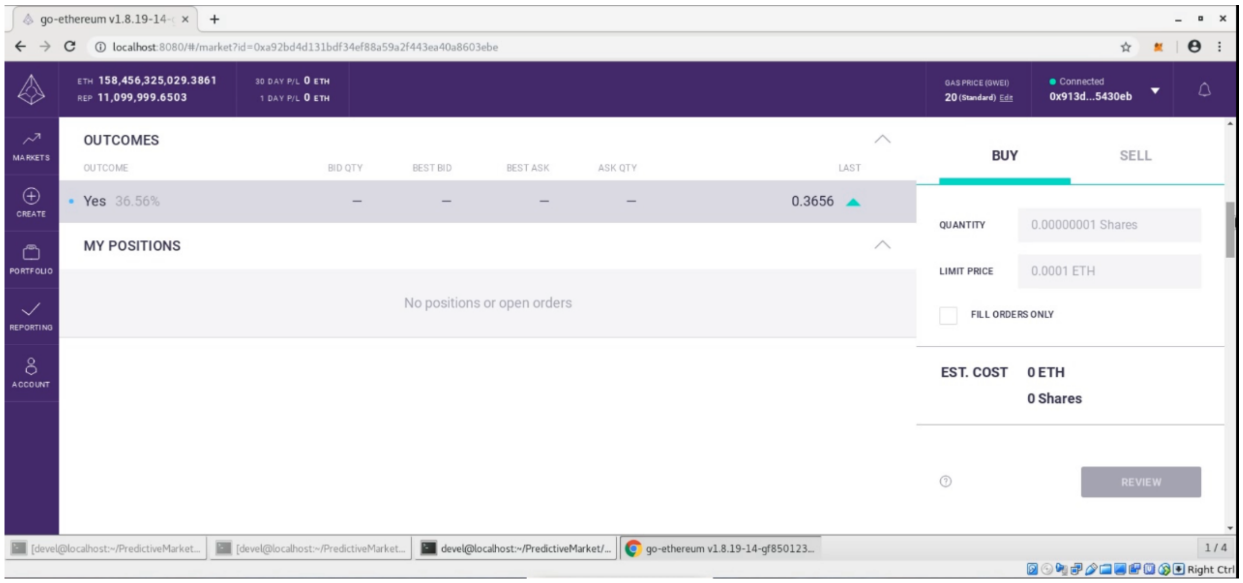
VI. Conclusion

The project introduces the novel idea of using predictive markets for software security using Augur. The existing Augur framework is extended by providing 1) a toolkit for the automatic creation of software security markets, 2) trading bots that simulate market activity, 3) modifications to Augur's UI that incorporate real-time information, and 4) a streamlined development environment. While these contributions are a substantial first step, the security framework has not yet been evaluated for effectiveness, as a large user base is required for an accurate test.

The market questions posed in Section V-A are by no means complete. A lawyer and an economist will be necessary to ensure the questions are precisely worded and provide no possibilities for loopholes or perverse incentives for malicious users. As a result, these questions are continually being refined. Future work will also need to address the issue of coordinated disclosure, and consider methods for using smart contracts to provide proof of bug disclosures. In the early stages of development, it's expected that these markets will operate on publicly disclosed bugs found before the market's strike date (time the market closes) that can be verified by the community.

Most importantly, this project requires and active user base in order to remain effective. The security provided by prediction markets is severely diminished when the number of participants drops. Early testing will only focus on versions of software that will interest a large sample of the target population. Once a sufficient user base has been established, longitudinal testing can begin, and the security of the software can be evaluated against the existing metrics mentioned in Section II.

While this work focused on using Augur for software security, it's possible that this framework could also be used in a number of other domains. Other domains include firmware security, hardware reliability, and internet of things (IoT) security. It should be noted that these ideas are presented merely as discussion points, and the legals and economic implications have not been thoroughly evaluated. The predictive market security platform could incentivize users to reverse engineer proprietary firmware and provide the results to the community. This would also allow the community to evaluate the security of the firmware, which is often much harder to analyze. The hardware reliability framework is the analog to the software security framework–it would allow users to speculate on the security and reliability of various hardware solutions and would address common hardware vulnerabilities that lead to exploits. Finally, IoT security would allow users to speculate on the security of various IoT devices. Hopefully, this would allow users to compare the relative security of comparable IoT devices and would show IoT developers that consumers are not satisfied buying insecure devices.



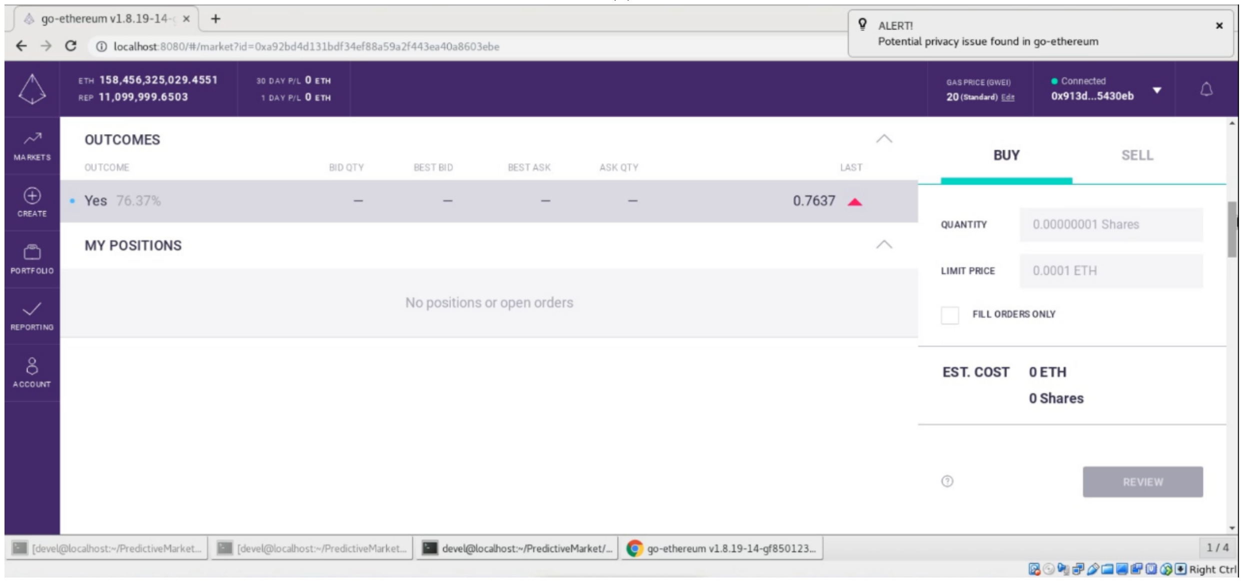
ETH 158,456,325,029.3861
REP 11,099,999.6503
30 DAY P/L 0 ETH
1 DAY P/L 0 ETH
20 (Standard) Edit
0x913d...5430eb

OUTCOME	BID QTY	BEST BID	BEST ASK	ASK QTY	LAST
Yes 36.56%	–	–	–	–	0.3656

No positions or open orders

QUANTITY 0.00000001 Shares
LIMIT PRICE 0.0001 ETH
EST. COST 0 ETH
0 Shares

(a)



ALERT!
Potential privacy issue found in go-ethereum

ETH 158,456,325,029.4551
REP 11,099,999.6503
30 DAY P/L 0 ETH
1 DAY P/L 0 ETH
20 (Standard) Edit
0x913d...5430eb

OUTCOME	BID QTY	BEST BID	BEST ASK	ASK QTY	LAST
Yes 76.37%	–	–	–	–	0.7637

No positions or open orders

QUANTITY 0.00000001 Shares
LIMIT PRICE 0.0001 ETH
EST. COST 0 ETH
0 Shares

(b)

Fig. 5: (5a) The probability of a privacy vulnerability being found in go-ethereum (v1.8.19-4) under normal market conditions. (5b) The probability of a privacy vulnerability being found in go-ethereum (v1.8.19-4) after a simulated alert discloses a potential bug.

References

[1] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[2] "The OWASP Foundation," https://www.owasp.org/index.php/Main_Page.

[3] Defense Security Cooperation Agency, "Security Assistance Management Manual," http://www.samm.dsca.mil/.

[4] U.S. Department of Commerce, "National Institute of Standards and Technology," https://www.nist.gov/.

[5] "Security standards in software development," https://www.kiuwan.com/blog/security-standards-in-software-development/.

[6] "Unanimous AI," Online, 2018, https://unanimous.ai/.

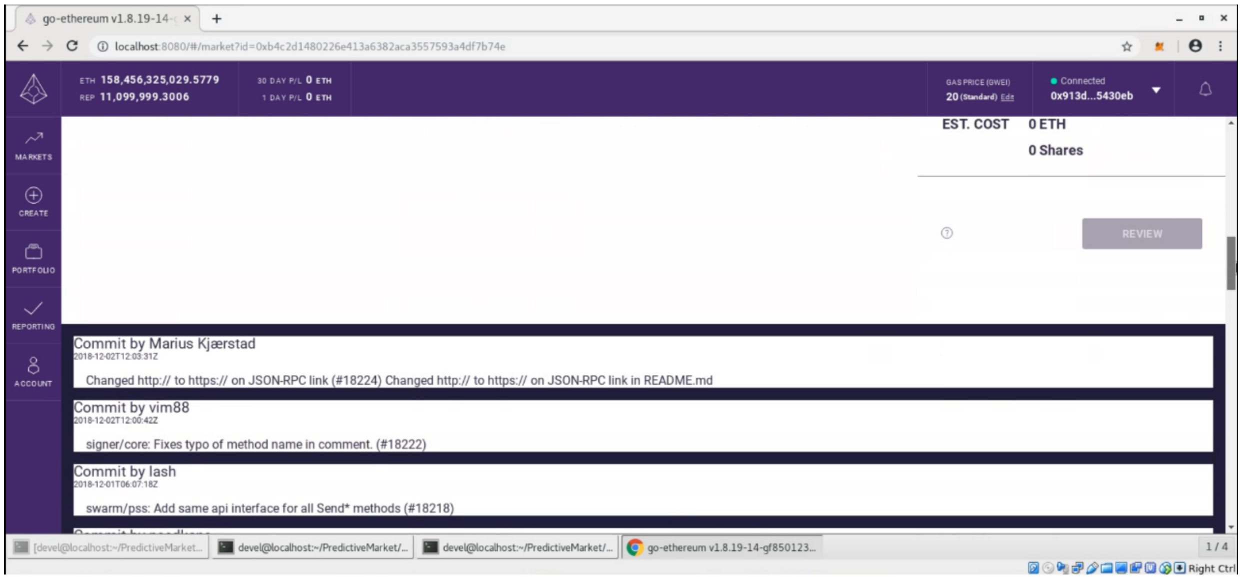
[7] L. Rosenberg and G. Willcox, "Artificial Swarm Intelligence vs Vegas Betting Markets," *IEEE Developments in eSystems Engineering*, 2018.

[8] A. Timmermann and C. W. Granger, "Efficient market hypothesis and forecasting," *International Journal of forecasting*, vol. 20, no. 1, pp. 15–27, 2004.

[9] R. Forsythe, F. Nelson, G. Neumann, and J. Wright, "The 1992 iowa political stock market: September forecasts," *The Political Methodologist*, vol. 5, no. 2, pp. 15–19, 1994.

[10] V. Buterin *et al.*, "Ethereum white paper, 2014," *URL https://github.com/ethereum/wiki/wiki/White-Paper*, 2013.

[11] C. J. Lee, "Whitepaper v2. 0 human consensus prediction."



Commit by Marius Kjærstad
2018-12-02T12:03:31Z

Changed http:// to https:// on JSON-RPC link (#18224) Changed http:// to https:// on JSON-RPC link in README.md

Commit by vim88
2018-12-02T12:00:42Z

signer/core: Fixes typo of method name in comment. (#18222)

Commit by lash
2018-12-01T06:07:18Z

swarm/pss: Add same api interface for all Send* methods (#18218)

Fig. 6: Example of a commit stream for a software security market.