A SYSTEM FOR PROFILING AND MONITORING DATABASE ACCESS PATTERNS BY APPLICATION PROGRAMS FOR ANOMALY DETECTION

NAVYA ALAPATI[1]

ANJANI VARMA CHINTALAPATI[2]

MR.ARIF MOHAMMAD ABDUL[3]

[1]Btech Student, Dept of CSE, GITAM University, Rudraram Mandal, Sangareddy district, Patancheru, Hyderabad, Telangana-502329, India.

[2]Btech Student, Dept of CSE, VNR Vignana Jyothi Institute of Engineering, Vignana Jyothi Nagar, Nizampet Rd, Pragathi Nagar, Hyderabad, Telangana-500090, India.

[3]Assistant Professor, Dept of CSE, GITAM University, Rudraram Mandal, Sangareddy district, Patancheru, Hyderabad, Telangana-502329, India.

ABSTRACT: Data base management systems provide access control systems that enable database managers (DBAs) to approve application programs access opportunities to data sources. Though such systems are effective, in method finer-grained gain access to control device tailored to the semiotics of the data kept in the DMBS is needed as a fabulous defense mechanism against smart assaulters. Hence, personalized composed applications which accessibility databases carry out an added layer of access control. For that reason, protecting a database alone is not nearly enough for such applications, as attackers targeting at taking information can capitalize on vulnerabilities in the blessed applications as well as make these applications to issue harmful database inquiries. An accessibility control device can only prevent application programs from accessing the data to which the programs are not licensed, yet it is not able to avoid misuse of the data to which application programs are licensed for accessibility. Hence, we require a mechanism able to find malicious behaviour arising from formerly accredited applications. In this paper, we provide the design of an anomaly detection mechanism, DetAnom that aims to fix such issue. Our strategy is based the evaluation and profiling of the application in order to create a concise depiction of its communication with the database. Such an account keeps a trademark for every sent question and likewise the equivalent constraints that the application program need to satisfy to send the inquiry. Later on, in the detection stage, whenever the application issues an inquiry, a component catches the query before it reaches the data source as well as validates the

matching signature as well as restraints against the existing context of the application. If there is an inequality, the query is marked as anomalous. The major benefit of our anomaly discovery system is that, in order to build the application profiles, we require neither any previous understanding of application susceptibilities nor any kind of instance of feasible attacks. As a result, our mechanism has the ability to secure the data from attacks customized to data source applications such as code alteration attacks, SQL injections, and likewise from various other data-centric strikes as well. We have applied our device with a software program testing technique called concolic screening as well as the PostgreSQL DBMS. Speculative results reveal that our profiling method is close to exact, needs appropriate quantity of time, and also the detection system sustains low run-time overhead.

Key Terms: Database, Insider Attacks, Anomaly Detection, Application Profile, SQL Injection.

I. INTRODUCTION

Data stored in databases is commonly essential to the company's operations as well as also sensitive, for example relative to personal privacy. Consequently, securing information stored in a database is an essential requirement. Data should be protected not just from external assaulters, however likewise from users within the organizations [3] A large range of establishments from government firms (e.g., armed forces, judiciary etc.) to business are seeing attacks by experts at a startling rate. One of the most crucial purpose of these experts is to either exfiltrate sensitive data (e.g., military plans, trade secrets, intellectual property, etc.) or maliciously changes the information for deceptiveness purposes or for attack preparation [1], [8] There are a number of truths that make the avoidance of expert assaults much more tough compared with various other standard (external) assaults [4] Initially, experts are allowed to accessibility sources, such as data and also computer system systems, as well as services inside the company networks as they have legitimate qualifications. Second, the actions of experts stem at a trusted domain name within the network, and thus are exempt to complete safety checks in the same way as exterior actions are. For instance, there is commonly no inner firewall program within the organization network. Third, experts are commonly very trained computer system professionals, that have expertise regarding the interior setup of the network and the protection and bookkeeping control deployed. Consequently, they might be able to

prevent traditional protection mechanisms. Shielding information from insider hazards needs incorporating various strategies. One crucial such method is stood for by the accessibility control system that is implemented as part of the data source management system code. A gain access to control system allows one to define which users/applications can accessibility which information for which objective. In addition to the gain access to control system carried out as part of the DBMS, applications might likewise execute their very own "application-level" gain access to control in order to carry out more complicated gain access to control plans. In such situations, accesses by individuals to the information stored in a database are mediated by the application programs. However, whereas the use of DBMS-level as well as application-level accessibility control systems provide an initial layer of defense versus expert dangers, these mechanisms are unable to safeguard versus destructive experts that have access to the applications as well as can thus change the code to transform the questions provided to the data source and also modify the logics of the application-level access control. Software-based attestation or easy stability measurement by a relied on platform component could be used for identifying any kind of unauthenticated adjustment to the application resource code by expert insiders. Nonetheless, attestation is commonly carried out throughout the loading of the application's executable as well as hence it cannot spot adjustments of program behaviours at run-time. Consequently, throughout implementation if a program is endangered by an insider utilizing known assault strategies, e.g., barrier overflow [9] or return-oriented programming attestation mechanisms cannot discover such harmful adjustments of behavior in the program. Likewise a destructive insider may be able to customize the information used for the attestation of the target application program, thus rendering attestation ineffective. Apart from that, utilizing simply a simple honesty measurement strategy is not a feasible service since this strategy cannot give integrity for self modifying code which is widely used as front end data source applications.

II. RELATED WORK

An official framework to categorize anomaly discovery systems has actually been suggested by Shu et al. According to this classification, our recommended strategy uses a deterministic language defined on the top of the database communications to do the detection. Numerous approaches have actually been suggested to safeguard databases against harmful application programs. DIDAFIT is

an intrusion detection system that works at the application degree. Like our system, DIDAFIT works in two stages: training stage and detection phase. Throughout the training phase, database logs are evaluated to create fingerprints of the inquiries found in the log. Fingerprints are routine expressions of inquiries with constants in the IN WHICH stipulation changed by place-holders that mirror the data kinds of the constants. During the detection stage, input queries are examined versus such finger prints. Queries that match some expression in the profiles are thought about benign, and also anomalous or else. DIDAFIT has nonetheless some major downsides. First, the system counts just on logs to develop program profiles. There is consequently no guarantee that the log would certainly include all legitimate inquiries. To address this downside, the authors recommend a strategy to create new trademarks from various other signatures that are similar in all sections as well as have some predicates in common. While this option operates in some instances, the system would not have the ability to identify questions that do not show up in the log. One more issue is that DIDAFIT does not think about the control flow and also information circulation of the program, i.e., the formula neither checks the appropriate order of the queries, neither the restraints that need to be confirmed for a question to be executed. The approaches suggested by Bertino et al. [5] as well as Valeur et al additionally evaluate training logs for developing accounts of questions. As a result they have the exact same drawbacks pointed out earlier. These approaches focus on the discovery of online attacks, like SQL Injection and Cross-Site Scripting (XSS) attacks, and also fall short to discover other assaults carried out via application programs, e.g., code alteration attacks. Safeguarding a database can be an uphill struggle, Paleari et al explained a brand-new group of assaults which depend on race problems. Such kind of attacks are less complicated in internet applications, where the tools made use of (primarily PHP and MySQL) provide a poor set of synchronization primitives however offer a highly identical setting. For that reason, when multiple at the same time requests are implemented, it is feasible to interleave the SQL queries in such a way that generates unforeseen habits. Such a sort of strike may be alleviated by a strategy, like the one we propose in this paper, which can impose the appropriate order of the inquiries. Our previous poster paper [27] describes some preliminary suggestions to shield versus data exfiltration with malicious adjustment of the application program. However, the approach recommended in this paper reduces the

efficiency expenses by allowing the ADE to simply go across the application account instead of concretizing of the symbolic execution tree of the application program. Such concretization in the discovery engine results in extra hold-up when validating a query. On top of that, our preliminary technique does not cover the combination of testing-based methods with program evaluation techniques nor cover execution and analysis of the recommended strategy.

III. PROPOSED MODEL

DETANOM ARCHITECTURE: The system architecture consists of several components, supporting the two phases of DetAnom, which we describe in what follows.

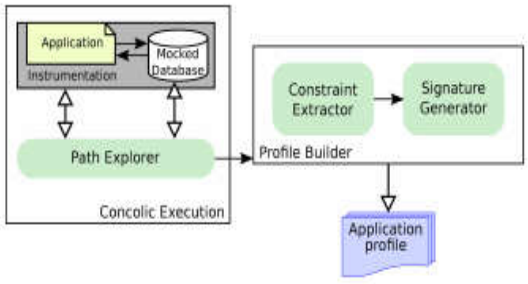

Fig 1: System architecture for profile creation

Number 1 reveals the components sustaining the profile production stage and their interactions. This phase begins by giving the application program as input to the concolic implementation module which first instruments the application. Note that the concolic execution does not require the application resource code. The bytecode is inspected using representation to discover the branches and track the input resources to the branch problems. After that, the application is started inside an instrumented digital machine which links the concolic execution engine to the networks utilized to connect with the individual. By doing this the concolic engine can generate input to compel the implementation of various branches. Consequently, the concolic implementation component implements the instrumented application for a variety of times with the purpose of exploring as several execution paths as feasible. Given that there is no warranty that the application terminates on each input, the concolic implementation utilizes a depth bounded search to restrict the profiling time. The depth of the search is a configurable parameter. Each time the application program concerns a question to the database, the restriction extractor in the account home builder component removes the restrictions that lead the application program to adhere to the present path. These constraints compose a part of the application account. In addition, each query sent to the data source is also sent to the account builder component where the

signature generator sub-module produces the signature of that query.

Anomaly Detection Component: The main modules supporting the anomaly detection phase are: the anomaly detection engine (ADE), the SQL proxy, the signature comparator, and the target database as shown in Figure 2.

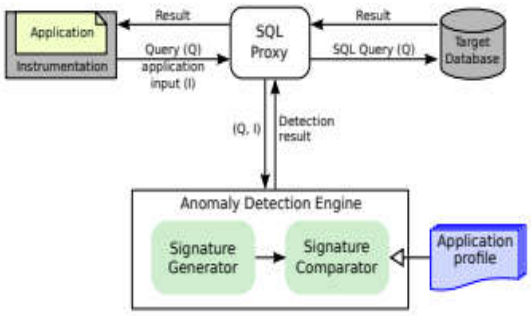

Fig 2: System architecture for anomaly detection

The information to protect is saved in the target database. We presume that the database web server is already protected to the very best of present security modern technology and can be accessed only through our proxy. The monitored application engages with the data source via SQL questions which are obstructed by the SQL proxy and also forwarded to the ADE for anomaly discovery. Additionally, the instrumented environment collects the application input and adds it as meta-data to the inquiry. The ADE additionally includes the trademark generator sub-module that produces the trademark of the received query. Upon getting the inquiry, the ADE checks whether the present program inputs please the restraints of some feasible implementation paths. If the restraints are completely satisfied, the trademark comparator contrasts the trademark of the inquiry connected with the pleased restraint to that of the received query. If there is a suit, the query is thought about legitimate, or else an anomaly is spotted. This information is then returned to the proxy, where a customized logic is utilized to determine the activities to be executed in order to take care of the anomaly. Instances of such activities consist of denying the inquiry, sending out an alarm system to a security manager, revoking the application program consents etc.

ADVERSARY MODEL: We presume that at run-time the application program can be tampered and hence end up being untrusted. As a result, we presume that while the program is performing, the program might issue a question that: (a) has actually never been come across in the profile production stage, i.e., the question does not come from the application at all; (b) belongs to the application yet is not pertinent to the current execution path; (c) pertains to the present execution course, yet the program input variables do not please that query's corresponding

restrictions. Every one of these instances can be easily mapped to well known protection attacks. In instance (a), an aggressor may just utilize a network sniffer or perform a man-in-the-middle attack to take the qualifications that the application utilizes to attach to the data source. Once the qualifications are swiped, the assailant might use any type of various other client to connect to the data source, avoid all the application degree safety checks, as well as concern inquiries that do not belong to the application. In case (b), an aggressor might acquire the qualifications as described in the previous situation and can utilize a similar method to tape-record the queries that the application problems. By repeating an allowed question the opponent can go through simpler safety checks and also therefore can obtain sensitive data. Let us think that a question gets only a row of sensitive data after the application has actually done some sanity checks on the values used to fetch the row. An assaulter may replay the query a number of times, changing only the worths made use of to filter the lead to order to retrieve all the data he/she wants. In instance (c), the aggressor jeopardizes the application and alters its gain access to control plan. For instance, most of the applications add an added layer of safety and security which requires the user to offer a pair of username and password. Typically, such applications recover a data source table for the given qualifications to recover the collection of permissions granted to the customer. Note that this level of protection is normally applied outside of the database. All the instances of the very same application make use of the same database qualifications for the connection and deal with the additional layer of safety and security inside. If an application is endangered so to return an effective authentication, on the database side we see only a sequence of enabled inquiries for which the constraints might not be pleased with the program inputs. We think that every component associated with the account development stage and also anomaly discovery stage is relied on. We likewise think that profiles are kept in a secure storage and also are not meddled by an expert or data source administrator.

The concolic implementation takes the application as input and instruments it to log each procedure that may impact a symbolic variable worth or a path problem. This component then performs the program concretely with some first default input. In order to explore other courses, it takes a look at the branch problems (i.e., restraints) along the carried out path, and also utilizes a constraint solver to find inputs that would turn around the branch problems. The execution is duplicated for

a number of times till all the implementation courses are checked out or the depth search limit is reached in all the explored ones. Taking into consideration that the goal of our system is to safeguard a database, we anticipate that the instrumented application problems inquiries along a few of these execution paths. The provided questions are sent to both the profile contractor and also the mocked data source. Upon getting a query, the constraint extractor sub-module in the profile home builder extracts the restraints that are prerequisite to carry out that query. The mocked data source uses the concolic engine to generate the question results that are needed to check out more recent execution paths.

Algorithm 1 Anomaly Detection

1: Input: Application Profile (AP)

2: vp = root of AP

3: while the program is executing do

4: q = issued query

5: ci = input constraints

6: signature generator generates sig(q)

7: found = false

8: for each child vi of vp do

9: if ci is satisfied then

10: signature comparator compares sig(q) to sig(queryi)

11: if signatures match then

12: response: NOT-ANOMALOUS

13: vp = vi

14: else

15: response: ANOMALOUS

16: end if

17: found = true

18: break

19: end if

20: end for

21: if found == false and vp is an incomplete node then

22: response: WARNING

23: end if

24: end while

Simple Detection: Instrumenting all the circumstances of the application to be safeguarded is not constantly feasible. Feasible reasons might relate, however not limited, to: - time restrictions as the application might be already deployed on a great deal of equipments and also upgrade every one of them may not be simple; - technical factors as the environment

utilized may not subject any type of API for the instrumentation (i.e. JVMs for mobile phones); - efficiency factors as in applications with high varieties of individual interactions, the overhead introduced by sending out the customer input may introduce substantial hold-ups. Consequently we have also developed a simple variation of our anomaly discovery strategy which does not need the instrumentation for the anomaly detection stage. We describe this strategy as simple detection, whereas we describe the previous technique as full detection. Since the profile creation stage does not transform, the very same account can be used for both the straightforward as well as complete detection stages. The main difference is that, without getting the application input during the anomaly detection, we can validate only that a permitted sequence of queries is released however without checking the restrictions.

IV. EXPERIMENTAL EVALUATION

We have actually examined the efficiency of our recommended DetAnom system. Our experiments have actually been performed on an online equipment running Ubuntu-14 as operating system, with 10GB of RAM memory and also 4 cpus. Thinking about the deterministic habits of our method, as well as thinking about that in case of a control-flow attack we expect to find all the queries after the attack to be flagged as anomalous, we focused the analysis on the efficiency as well as the overhead required to send the user input as well as validate the restraints. Since to the very best of our knowledge there is no public offered dataset appropriate for our requirements, we produced some test applications. The objective was to examine DetAnom utilizing applications with various sizes, in order to examine the actions in instance of partial accounts. Gotten by raising complexity; the first 2 usage just binary branches, while the 3rd has additionally for loops. As we can see in the 2nd column, the account creation time boosts very fast. The reason is that in the most awful instance this moment is exponential in the variety of branches. A restriction of the concolic screening tool we use is that the backtrack support is not implemented. Consequently each time a new branch needs to be checked out, a new execution of the application is needed. Considering that we produced the test applications nesting binary branches equally, profiling an application with added “if-else” calls for two times the moment. Including loopholes slows down a lot more the account creation since, as described in Area 5, jCute actually unroll loops that can be seen as a collection of nested "if" s where every "if", but the last

one, contains the loop body and also the following if. To check the applications, a pseudo random input generator has been utilized to imitate the individual input. Booting up the generator with the exact same seed makes it possible to test the exact same implementation circulation. We examined 100 various implementation streams for each application. For each execution flow we tape-recorded the execution time and also the network use of the application in both a typical implementation and an implementation secured by DetAnom. As we can see the runtime expenses is small and around 20%. We can additionally see that the ordinary execution time of the longer application is just few nanoseconds more than the implementation time of the smaller one. The reason is that the time required to begin the JVM is substantially higher than the moment required sending out a query

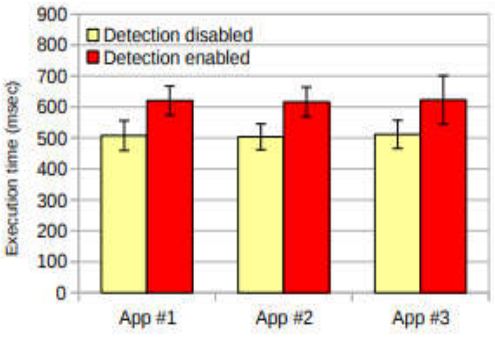

Fig 3: Execution time overhead

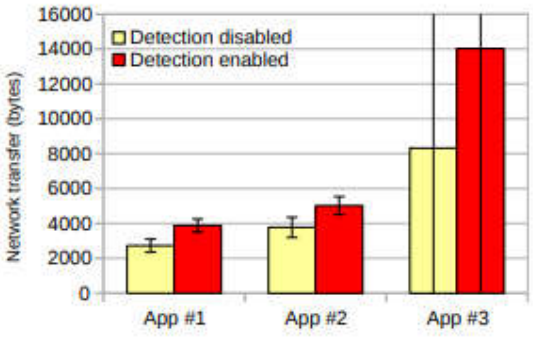

Fig 4: Network overhead

Our experiments have actually revealed some technological restrictions of our current strategy. In what follows we talk about such constraints as well as overview possible options. The profile creation stage is really sluggish. This is a restriction of the screening strategy we make use of which really runs the program as often times as it is needed to check out the possible implementation courses. In addition, the concolic testing device we made use of, JCute [30], was developed to compose small unit tests. For that reason it does not implement any kind of device to accelerate the evaluation of large programs. Numerous executions could be parallelized and distributed on various equipments; in addition, saving a snapshot of the execution in order to having the ability to backtrack without the demand of reactivating the application from the get go might result in a big enhancement of the profile development time. Using a concolic engine, which sustains backtracking, doing

photo of the application implementation, might additionally serve in supporting the incremental account production. This can be made use of both to promptly release a partial account to start safeguarding the application while constructing a much more accurate account, or to incrementally transform the profile to mirror application updates.

V. CONCLUSION AND FUTURE WORK

Though accessibility control devices released in DBMS have the ability to avoid application programs from accessing the information for which they are not licensed, they are incapable to prevent data abuse triggered by authorized application programs. In this paper, we have actually suggested an anomaly discovery mechanism that has the ability to recognize anomalous queries resulting from formerly accredited applications. Our mechanism builds close to precise account of the application program, without the need of its source code, as well as checks at run-time inbound queries versus that profile. Along with anomaly discovery, our DetAnom system is capable of finding any kind of injections or adjustments to the SQL queries. We wish to emphasize 2 benefits of our strategy contrasted to other much more conventional techniques. The very first is that by using the concolic testing strategy instead of static analysis strategies, we can profile the actual implementation of the code that includes inquiries carried out by self-modifying or dynamically downloaded and install code. The second is that we have the ability to impose the real order of the questions sent to the database, unlike conventional SQL injection discovery techniques which are unable to identify whether a query is added or gotten rid of from an application program. We have applied DetAnom with JCute and also PostgreSQL which leads to reduced run-time overhead and also high accuracy in detecting strange database accessibilities. We are currently prolonging our job along a number of directions. Our existing application of DetAnom exploits the restraints that JCute [30] assistances, i.e., arithmetic, pointer, as well as thread restrictions. We plan to improve our trademark generation system by incorporating details concerning program constants, variables, sensible as well as relational operators utilized in the IN WHICH clause of an inquiry as this information might enhance the precision of discovery. We likewise plan to improve the completeness and precision of our account production device utilizing both fixed and dynamic evaluation of the program. In this strategy, we will first examine the program statically to locate all the implementation paths which contain

SQL queries and after that guide the concolic implementation dynamically to ensure that it does not leave any courses unexplored.

VI. REFERENCES

[1] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. Commun. ACM, 56(2):82–90, Feb. 2013.

[2] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. VLDB Endow., 5(11):1471– 1482, July 2012.

[3] M. Collins, D. M. Cappelli, T. Caron, R. F. Trzeciak, and A. P. Moore. Spotlight on: Programmers as malicious insiders (updated and revised). Technical report, Carnegie Mellon University, 2013. http://resources.sei.cmu.edu/asset files/WhitePaper/ 2013 019 001 85232.pdf.

[4] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference and Exposition, 2000. DISCEX ’00. Proceedings, volume 2, pages 119–129 vol.2, 2000.

[5] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE ’09, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.

[6] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA ’07, pages 151–162, New York, NY, USA, 2007. ACM.

[7] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS ’04, pages 318–329, New York, NY, USA, 2004. ACM.

[8] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In Proceedings of the 11th Annual Network and Distributed System Security Symposium NDSS, 2004.

[9] W. G. Halfond, J. Viegas, and A. Orso. A classification of sqlinjection attacks and countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering, volume 1, pages 13–15. IEEE, 2006.

[10] S. R. Hussain, A. M. Sallam, and E. Bertino. Detanom: Detecting anomalous

database transactions by insiders. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pages 25–35. ACM, 2015.

[11] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. CS701 Construction of compilers, 19, 2005.

[12] E. Bertino. Data Protection from Insider Threats. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael, 2012.

[13] E. Bertino and G. Ghinita. Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 10–19, New York, NY, USA, 2011. ACM.

[14] E. Bertino, A. Kamra, and J. P. Early. Profiling database application to detect sql injection attacks. In IEEE International Performance, Computing, and Communications Conference, IPCCC 2007, pages 449– 458, April 2007.

[15] R. Majumdar and K. Sen. Hybrid concolic testing. In Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pages 416–426, May 2007.

Navya Alapati: Btech Student, Dept of CSE, GITAM University, Rudraram Mandal, Sangareddy district, Patancheru, Hyderabad, Telangana-502329, India.

Anjani Varma Chintalapati: Btech Student, Dept of CSE, VNR Vignana Jyothi Institute of Engineering, Vignana Jyothi Nagar, Nizampet Rd, Pragathi Nagar, Hyderabad, Telangana-500090, India.

Mr Arif Mohammad Abdul: Assistant Professor, Dept of CSE, GITAM University, Rudraram Mandal, Sangareddy district, Patancheru, Hyderabad, Telangana-502329, India.