Noverdose

Team HackNah
SeoYeong Lee, Sihyun Park, Nicole Leung, TaeHyeon Park, Kimberly Hope Jackson

---

1. Introduction

Noverdose aims to provide simple and reliable content to individuals seeking accurate information on the potential interactions and risks when consuming two different drugs at once. By simplifying the decision-making process of drug intake, our application aspires to enhance user safety and promote a more neat approach to medication management.

---

2. Noverdose GitHub Repository Link

The following is the link to our project, Noverdose GitHub repository:
https://github.com/maroo1746/noverdose.git

3. Project Overview

Here is an overview of our project development process and implemented features:

a. Development Process

Our project development environment is as follows:

- Framework: Django.
- Front-End: HTML, CSS.
- Back-End: Python, C.
- Database: MySQL.
- Containerization: Docker.

First, we created a MySQL database by digitizing the list of medicines and the list of contraindicated medicines provided by the government - HRLA Bigdata Open portal. (https://opendata.hira.or.kr/op/opc/selectOpenDataList.do?pageIndex=5) During this process, unnecessary data was removed, and the structure of the database was established.
Subsequently, we concurrently worked on a simple front-end and back-end to implement a medicine search feature. We also built a users database to allow for the addition of medicines when accessed by an admin account.
Finally, based on a team member's suggestion, we implemented a feature allowing users to save the medications they are taking on their personal page and check at once if there is any contraindicated combination among all the medicines they are taking.

The key tables in the database are as follows:

- med table stores drug information. Figure 3.1 shows a snippet of the med table. Regarding the medicines, a drug was considered different if it did not match in at least one of the five columns: compound_name, compound_code, product_code, product_name, or company_name.

id	compound_name	compound_code	product_code	product_name	company_name
1	aceclofenac	100901ACH	648101510	에이서캡슐(아세클로페낙)_(0.1g/1캡슐)	경동제약(주)
2	aceclofenac	100901ACS	644304150	클란자에스연질캡슐(아세클로페낙)(수출명:Sp...	한국유나이티드제약(주)
3	aceclofenac	100901ATB	52400690	로페낙정(아세클로페낙)_(0.1g/1정)	(주)인트로바이오파마
4	aceclofenac	100901ATB	52700370	케페낙정100밀리그램(아세클로페낙)_(0.1g/1정)	크리스탈생명과학(주)
5	aceclofenac	100901ATB	53300200	아세크릴정(아세클로페낙)_(0.1g/1정)	(주)중헌제약

Figure 3.1 med_db.med table.

- medcontraindication table stores combinations of drugs that are contraindicated by storing two medicine's id together in one row. Figure 3.2 shows a snippet of the medcontraindication table.

id	med_id_a	med_id_b	contraindicated_info	notification_no	notification_date	detail_info
1	1	6257	중증의 위장관계 이상반응	20050017	2005-03-09	NULL
2	1	6258	중증의 위장관계 이상반응	20050017	2005-03-09	NULL
3	1	6259	중증의 위장관계 이상반응	20050017	2005-03-09	NULL
4	1	6260	중증의 위장관계 이상반응	20050017	2005-03-09	NULL
5	1	6261	중증의 위장관계 이상반응	20050017	2005-03-09	NULL

Figure 3.2 med_db.medcontraindication table

- searchmed_usermedication table stores the combination of the user and the drugs they use. When a user clicks the search button, the backend function searches for combinations of med_ids that match the current logged-in user's user_id and identifies any contraindicated combinations. Figure 3.3 shows a snippet of the searchmed_usermedication table.

id	med_id	user_id
2	10	2
3	4	2
6	194	2
7	583	2
8	616	2
9	536	2
10	121	2
11	6257	2
12	2358	2
13	6257	6
15	4	6
16	10	6
17	119	6
18	199	6
19	381	6
20	6245	6
22	6257	7
23	21	7
24	667	7
25	381	7

Figure 3.3 med_db.searchmed_usermedication table.

The workflow for each function is as follows:

- Initially, the HTML structure and database design were developed concurrently.
- Subsequently, the backend team enabled database information to be displayed on the web page.
- In brief, a client sends a JSON AJAX request to the Django server.
- The Django server, linked to the MySQL database, fetches data and responds in JSON.
- This response is then displayed on the client's web page.

Refer to Figure 3.4 to further facilitate the workflow explanation.

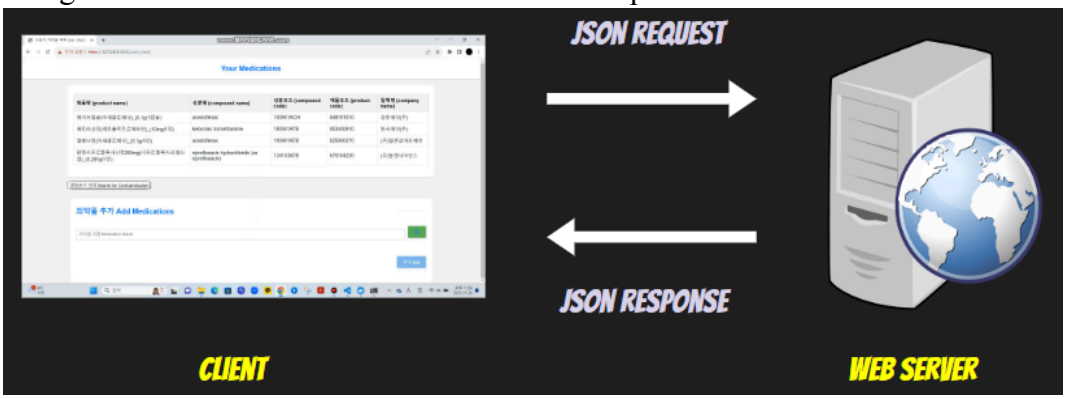


Figure 3.4 Workflow for each function.

b. Implemented Features

The features implemented in our project are as follows:

- **Medication Contraindication Search:** By pressing the "Contraindications for Combined Usage" button, the user will be taken to the contraindication search page. Any user, no matter their status, can access this page. The search page prompts the user for two medications that they intend to take together. After selecting their medications from a dropdown menu, the user can press the search button in the bottom right corner. Consequently, a table will appear with contraindication information if the two medications are not safe to take together.
- **Admin Account:** A user is considered "Admin" if they have been given root privilege to manipulate the database. By selecting the "Admin Login and Adding Info" option, the user will be taken to a login page prompting for a username and password. After authenticating the user, the website will ask for information about the medication that will be added to the database. By selecting "submit" after accurately adding the required information, the new medication will be added to the application's database and the log is recorded in the 'log.txt' file.
- **Creating a General User Account:** By selecting "User Registration," the user will be taken to an initial login page that will guide them through the creation of a general user account, prompting for a sufficient username and password.
- **General User Account:** By following the "Normal User Login" the user will be authenticated and given the opportunity to choose to use the contraindication function (described previously), or register the current medications they are taking. In this page the user can add medications under the "Add Medications" prompt, where their chosen medications will be displayed in a table labeled "Your Medications." By pressing the "Search for Contraindication" option underneath the aforementioned table, the application will check if any of the user's medications are unsafe to take together. If there is a contraindication, the information will be displayed in a new table below the "Search for Contraindication" button.

The link to our website's demo video is as follows: https://youtu.be/OtbAoTTrhQo

4. GitCTF Results

The results of our team's GitCTF experience are as follows:

4.1. Intended Vulnerabilities

Noverdose

Our project contains several vulnerabilities all located in the med_db.c file. Note that all of them have been patched in the final submission. The intended vulnerabilities are as follows:

- **Buffer Overflow Attack (BOF):** Since 'sprintf' does not check the size of the buffer, if the input is too large, it can overflow the query or log_message buffers. This could open up the possibility for various attacks, including arbitrary code execution.
- **SQL Injection Attack:** The program constructs query strings with the 'sprintf' function, including unvalidated user inputs. This can lead to SQL injection if the inputs contain malicious SQL commands, risking compromising our database.

```
char query[500];
sprintf(query, "INSERT INTO med(product_name, compound_name, compound_code, product_code, company_name)
VALUES('%s', '%s', '%s', '%s', '%s')", product_name, compound_name, compound_code, product_code, company_name);
```

Figure 4.1.1 Location of BOF and SQL injection vulnerabilities in med_db.c.

- **Format String Attack:** The call 'fprintf(stderr, product_name);' uses 'product_name' as a format string. If this string contains format specifiers like '%s', it could lead to unexpected behavior.

```
sprintf(log_message, "Inserted: %s, %s, %s, %s, %s", product_name, compound_name, compound_code, product_code, company_name);
write_to_log(log_message);
```

Figure 4.1.2 Location of format string vulnerability in med_db.c.

- **Error-Based Attack:** Our program lacked robust exception handling. It just prints an error and exits if the database connection fails or a query issue happens. This exposes it to error-based attacks, where detailed error messages are exploited to understand the database structure, potentially leading to further system exploits.

```
if (mysql_query(con, query)) {
    fprintf(stderr, "%s\n", mysql_error(con));
    mysql_close(con);
    return;
}
```

Figure 4.1.3 Location error-based vulnerability in med_db.c.

- **Race Condition Attack:** The potential race condition related to the log.txt file in the code arises from the fact that there is a time gap between when the program checks for the existence of the file (or opens it for appending) and when it actually writes to it. An attacker could exploit this time gap in what is known as a "Time of Check to Time of Use" (TOCTOU) race condition.

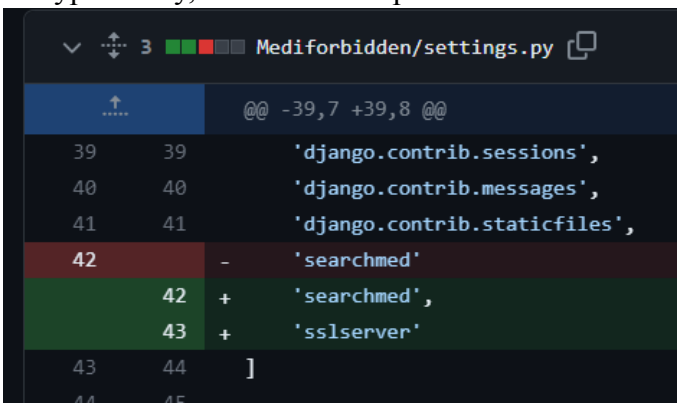
```c
void write_to_log(const char* message) {
    int fd = open(LOG_FILE, O_WRONLY | O_CREAT | O_APPEND, LOG_FILE_MODE);
    if (fd == -1) {
        perror("Failed to open log file");
        return;
    }

    // fd를 FILE*로 변환합니다.
    FILE* fp = fdopen(fd, "a");
    if (!fp) {
        perror("Failed to convert file descriptor to file pointer");
        close(fd);
        return;
    }

    fprintf(fp, "%s\n", message);
    fclose(fp); // fclose는 내부적으로 fd도 닫습니다.
}
```

Figure 4.1.4 Location of race condition vulnerability in med_db.c.

4.2. Unintended Vulnerabilities

Here are the unintended vulnerabilities we received from other teams. A total of three unintended vulnerabilities:

a. **Unencrypted Packet Sniffing (reported by Team 4):** This vulnerability comes from the login process where sensitive information like passwords goes through transmission in the form of plain text. The patch we made was to use HTTPS for entire packet encryption. HTTPS, or Hypertext Transfer Protocol Secure, protects the transmission of data by establishing an encrypted link between the client and the server. This encryption is achieved through the use of SSL/TLS protocols, which create a secure channel over an insecure network. This means that when a user inputs their password, it's encrypted right from their device and remains so until it reaches the server. Only the intended server, with the correct decryption key, can access the password's content.

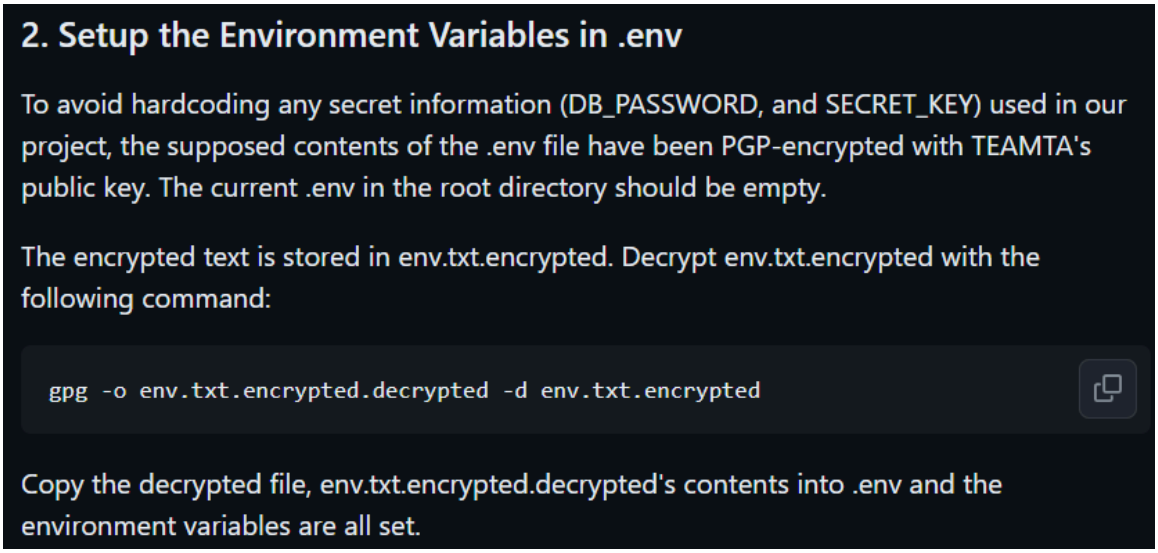


To achieve this, we replaced our existing web server "runserver" with "sslserver."

b. **Buffer Overflow (reported by Team 8):** ‘product_name’, ‘medA_id’, and ‘medB_id’ are not validated before being used to access the database. There could be a type check before being used to request from the database to stop empty strings or add a validation check. The patch we made was to implement an input validation, use parameterized queries, and use Q Objects. Q Objects are used to express complex query conditions in Django for dynamically constructing conditions. The relevant code is as follows:

```
34 +     if medA_id and medB_id:
35 +         results = Medcontraindication.objects.filter(
36 +             Q(med_id_a=medA_id, med_id_b=medB_id) | Q(med_id_a=medB_id, med_id_b=medA_id)
37 +         )
38 +     else:
39 +         results = Medcontraindication.objects.none()
```

c. **Password Guessing (reported by Team 1):** This vulnerability is rooted in the mysql_real_connect password being hardcoded. However, for a MySQL database password, the application needs to use the original password to establish a connection to the database. If the password were hashed, it would be impossible to retrieve the original password from the hash, making it impossible to use for database authentication. Thus, the patch we made was storing it in a hidden file (.env) as an environment variable and encrypting that file with TEAMTA's public key. If the TA(or any other third parties that have an authorization to access the .env file) uses his own secret key, he will be able to get the original root user password of the MySQL database safely and connection between database process and web process will be established. The picture below is part of the project's README.md file. It indicates that we have encrypted the .env file with the public key of a user authorized to access the environment variable file (here, TEAMTA's public PGP key), and instructs to decrypt it back to use the environment variables.



Additionally, if individuals in the group authorized to share environment variables generate a new public/private key pair at regular intervals and submit their public keys to the project's main administrator (which is now us), we can enhance security by encrypting the '.env' file with the new key pair. Utilizing tools like GnuPG or OpenPGP.js to implement an automated process for regularly updating PGP (Pretty Good Privacy) keys will allow for more secure storage of the environment variable files.

4.3. Vulnerabilities Reported

Noverdose

Here are all the vulnerabilities we found on other teams' projects and reported them. A total of 21 valid vulnerabilities. These are the vulnerabilities reported for:

a. Team 1's (4 vulnerabilities):

- **Format String:** This vulnerability occurs when the program doesn't properly validate the submitted input. In this case, the 'fprintf' in 'saveProfQuiz.c' file, 'save_quiz()' function expects more arguments as input, and if these arguments are not supplied, the function could read or write the stack.
- **Buffer Overflow:** This vulnerability comes from using 'strcpy' in 'saveQuizSubmit.c' file, 'load_to_lecture()' function, which fails to check the length of its arguments. This means that an attacker can overwrite parts of the program using the source string variable, 'submit_text'.
- **Buffer Overflow:** This vulnerability also comes from using 'strcpy' in 'saveNotice.c' file, 'save_notice()' function. It is the same case as the previous BOF.
- **Misconfiguration:** This vulnerability comes from creating a file based solely on the name of the lecture, and not checking the owner of the file. In a user scenario assume ProfessorA and ProfessorB both have a lecture named 'Science'. StudentA signs up for the Science lecture taught by ProfessorA. If the student submits a quiz for that lecture, their submission will go to all lectures named 'Science' regardless of professor or problem question.

b. Team 3's (5 vulnerabilities):

- **Buffer Overflow:** The function 'validate()' in the 'passwordValidator.cc' file contains a buffer overflow vulnerability. It uses 'strcpy' to copy the input string into a buffer of a fixed size (BUF_SIZE which is 100) without checking if the input string is longer than the buffer. This can lead to a buffer overflow if the input string exceeds 100 characters.
- **Buffer Overflow:** The function 'validate()' in the 'usernameValidator.cc' file contains a buffer overflow vulnerability. It is the same case as the previous BOF.
- **Buffer Overflow:** The function 'validate()' in the 'majorValidator.cc' file contains a buffer overflow vulnerability. It is the same case as the previous BOF.
- **Buffer Overflow:** The function 'validate()' in the 'emailValidator.cc' file contains a buffer overflow vulnerability. It is the same case as the previous BOF.
- **Buffer Overflow:** The function 'validate()' in the 'majorValidator.cc' file contains a buffer overflow vulnerability. It is the same case as the previous BOF.

c. Team 4's (1 vulnerability):

- **Potential Timing Attack Vulnerability:** In the code, the line if (login_hash == account_pass.hash) compares the computed hash of the user-provided password with the hash stored in the database. The standard string comparison operation (==) may take slightly different amounts of time depending on how many characters in the strings match before a mismatch is found. An attacker could potentially use this timing information to infer the hash, character by character.

d. Team 5's (6 vulnerabilities):

- **SQL Injection:** The code in 'login.c' file constructs SQL queries by directly embedding user inputs (id, pw) into SQL statements (e.g. "SELECT * FROM USERS WHERE id = 'user_id'). This approach is vulnerable to SQL injection attacks. The following is a link to the SQL Injection Attack execution video: https://youtu.be/1slsa_rjajc?si=_Hta0YBJOokDMzyI

Noverdose

- **Integer Overflow:** There is a risk of integer overflow in ‘game3.c’ file when ‘user_bet’ is multiplied by (user_ans + 1). If ‘user_bet’ is a large integer, this multiplication could overflow, leading to unexpected and incorrect results. The following is a link to the Integer Overflow Attack execution video: https://youtu.be/6wKU8BrlrA4?si=R09mmXghC2HlhQI2
- **Command Injection:** The program uses user-provided input for the article titles in ‘board.c’ file to construct file paths and names without sufficient validation or sanitization. This could allow a user to execute arbitrary commands on the program.
- **Format String:** The ‘printf’ function in ‘game1.c’ file is used with the 'money' variable directly as the format string. Since 'money' is inputted with user input from the ‘fgets’ function, it may contain format specifiers (e.g. %s, %d, %x, etc.). The ‘printf’ function will interpret these format specifiers and try to access the corresponding variables or memory locations, which are not provided.
- **Buffer Overflow:** The ‘gets’ function in ‘game4.c’ file is used to read input into the input buffer. 'gets' function continues to read input until it encounters a newline (\n) or end-of-file (EOF) character. It it does not perform bounds checking on the input size.
- **Buffer Overflow:** The buffer in ‘game2.c’ file is declared with a size of 8 bytes (char buffer[8];), but ‘memcpy(buffer, input, 199);’ is used to copy 199 bytes from input into buffer. Here, ‘memcpy()’ does not perform any bounds checking and will copy the specified number of bytes regardless of the destination buffer size.

e. Team 8’s (5 vulnerabilities):

- **Lack of Rate Limiting and Account Lockout Mechanisms**: The application handles user logins by checking credentials against the database. However, there are no mechanisms in place to limit the number of attempts a user (or an attacker) can make to log in. This absence makes the application susceptible to brute-force attacks, where an attacker tries many password combinations until they find the right one.
- **Insecure Direct Object References (IDOR) / Broken Access Control:** The application allows unauthorized access to sensitive pages/functions (/purchase, /updateEmail) that should typically require user authentication. The lack of proper session management and access control checks means that an attacker can simply type in the URL of these pages and gain access, potentially leading to unauthorized actions, data exposure, or data manipulation.
- **Buffer Overflow:** The get_user_input function in the provided code is vulnerable to a buffer overflow due to it using a fixed buffer size and having uncontrolled input. The function uses ‘fgets’ to read input into these fixed-size buffers without verifying if the input size exceeds the buffer's capacity. If the user input is longer than the buffer size, it will lead to a buffer overflow, as ‘fgets’ does not automatically handle buffer size exceeding the predefined limit.
- **Empty Password Vulnerability:** The code in ‘src/mongo.js’ file defines a schema for user login in a MongoDB database using Mongoose, which potentially has a vulnerability related to allowing empty passwords due to the way the password field is defined in the ‘logInSchema’. In the ‘logInSchema’, the password field is defined with ‘required: false’. This means that the schema does not enforce the requirement of a password when creating or updating a document in the LogInCollection. As a result, it's possible to create a user record with an empty or null password. Allowing empty passwords can be a significant security risk. It means that users (or records) can be created without any password, making it trivial for unauthorized users to gain access by simply leaving the password field empty during login.

- **Format String Vulnerability:** The function in ‘inputCheck.c’ file tries to write to a file using ‘printf(file, "%s,%s\n", user->name, user->password);’. However, ‘printf’ is not designed to write to files; it is intended for standard output. The correct function to write formatted strings to a file is ‘fprintf’.

5. Lessons Learned during GitCTF

We learned a lot during GitCTF. These are each member's thoughts about what we got to learn and experience during GitCTF.

a. **SeoYeong’s:** I tried to develop without as many vulnerabilities as possible, but other students found vulnerabilities that l had not noticed. I don't think I could have found these vulnerabilities on my own. So I realized the importance of cross-checking, and it was a very valuable experience.

b. **Nicole’s:** Throughout my experience with GitCTF I gained a lot of knowledge about both web development and software security. Although the task of building the application and inserting the vulnerabilities was very difficult and daunting, I’m glad I got to work with my teammates and discuss the issues together.

c. **Sihyun’s:** I think it was a good learning experience, I quite enjoyed the collaborative aspect of doing a project in a class environment. It was my first time working with django and I am quite happy that through the project I was able to understand how the framework works more clearly.

d. **Kimberly’s:** It has been a great and challenging experience to make this project alongside my team members and I learned a lot from both the development process and through GitCTF. I hope we will be able to apply all that we learned here in future project developments, especially in terms of enhancing their security and also in pursuing our prospective careers.