Software Security: Reasoning About Code

CS 161: Computer Security

Prof. David Wagner

January 27, 2016

```
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
         total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
     size(a) >= n &&
          ???                        */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
         total += *(a[i]);
    return total;
}
```

```c
/* requires: a != NULL &&
     size(a) >= n &&
     for all j in 0..n-1, a[j] != NULL */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```c
char *tbl[N]; /* N > 0, has type int */


int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct postcondition for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                       /* 0 <= h */
  while (*s)                        /* 0 <= h */
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                     /* 0 <= h */
  while (*s)                      /* 0 <= h */
    h = 257*h + (*s++) + 3;       /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}
```

Is the postcondition correct?
(a) Yes, (b) 0 <= retval is correct,
(c) retval < N is correct, (d) both are wrong.

```c
                                  0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                   /* 0 <= h */
  while (*s)                    /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                    /* 0 <= h */
  while (*s)                     /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                     /* 0 <= h */
  while (*s)                      /* 0 <= h */
    h = 257*h + (*s++) + 3;       /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                   /* 0 <= h */
  while (*s)                    /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}
```

What is the correct postcondition for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                      /* 0 <= h */
  while (*s)                       /* 0 <= h */
    h = 257*h + (*s++) + 3;        /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

Fix?

```c
char *tbl[N];

/* ensures: 0 <= retval && retval < N */
unsigned int hash(char *s) {
  unsigned int h = 17;          /* 0 <= h */
  while (*s)                    /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  unsigned int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

Common Coding Errors

- Memory safety vulnerabilities
- Input validation vulnerabilities
- Time-of-Check to Time-of-Use (TOCTTOU) vulnerability (later)

Input Validation Vulnerabilities

- Program requires certain assumptions on inputs to run properly
- Programmer forgets to check inputs are valid => program gets exploited
- Example:
  - Bank money transfer: Check that amount to be transferred is non-negative and no larger than payer’s current balance

Access Control and OS Security

CS 161: Computer Security

Prof. David Wagner

January 27, 2016

Types of Security Properties

- Confidentiality
- Integrity
- Availability

Access Control

- Some resources (files, web pages, …) are sensitive.
- How do we limit who can access them?

- This is called the *access control* problem

Access Control Fundamentals

- *Subject* = a user, process, …
(someone who is accessing resources)
- *Object* = a file, device, web page, …
(a resource that can be accessed)
- *Policy* = the restrictions we'll enforce

- *access*(S, O) = true
if subject S is allowed to access object O

Example

- *access*(Alice, Alice's wall) = true
  access(Alice, Bob's wall) = true
  access(Alice, Charlie's wall) = false

- *access*(daw, /home/cs161/gradebook) = true
  access(Alice, /home/cs161/gradebook) = false

Access Control Matrix

- *access*(*S*, *O*) = true
  if subject *S* is allowed to access object *O*

	Alice's wall	**Bob's wall**	**Charlie's wall**	**…**
Alice	true	true	false	
Bob	false	true	false	
…				

Permissions

- We can have finer-grained permissions, e.g., read, write, execute.

- *access*(daw, /cs161/grades/alice) = {read, write}
  access(alice, /cs161/grades/alice) = {read}
  access(bob, /cs161/grades/alice) = {}

	/cs161/grades/alice
daw	read, write
alice	read
bob	-

Access Control

- Authorization: who *should* be able to perform which actions
- Authentication: verifying who is requesting the action

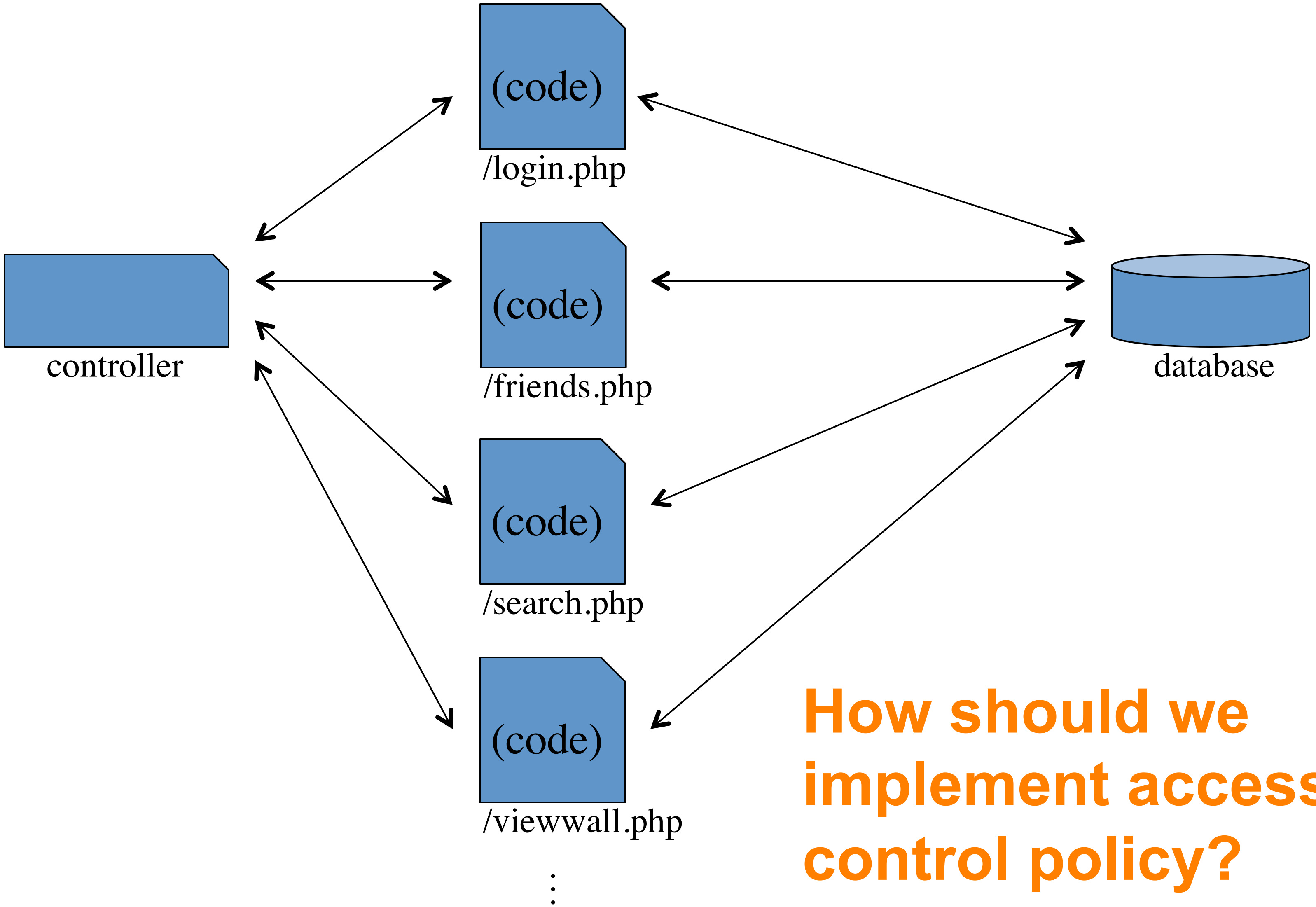
Access Control

- Authorization: who *should* be able to perform which actions
- Authentication: verifying who is requesting the action
- Audit: a log of all actions, attributed to a particular principal
- Accountability: hold people legally responsible for actions they take.

Web security

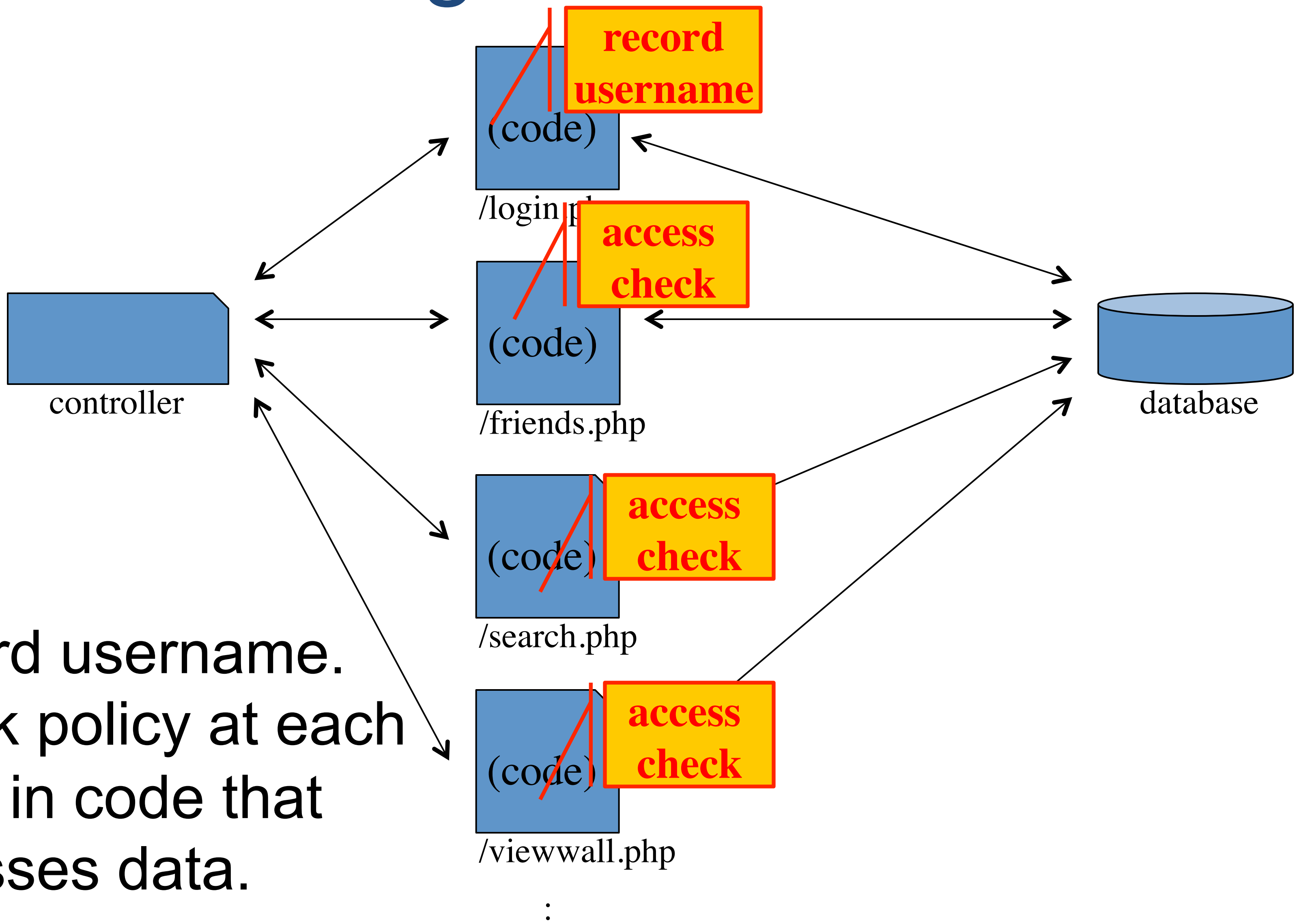
- Let’s talk about how this applies to web security…

Structure of a web application



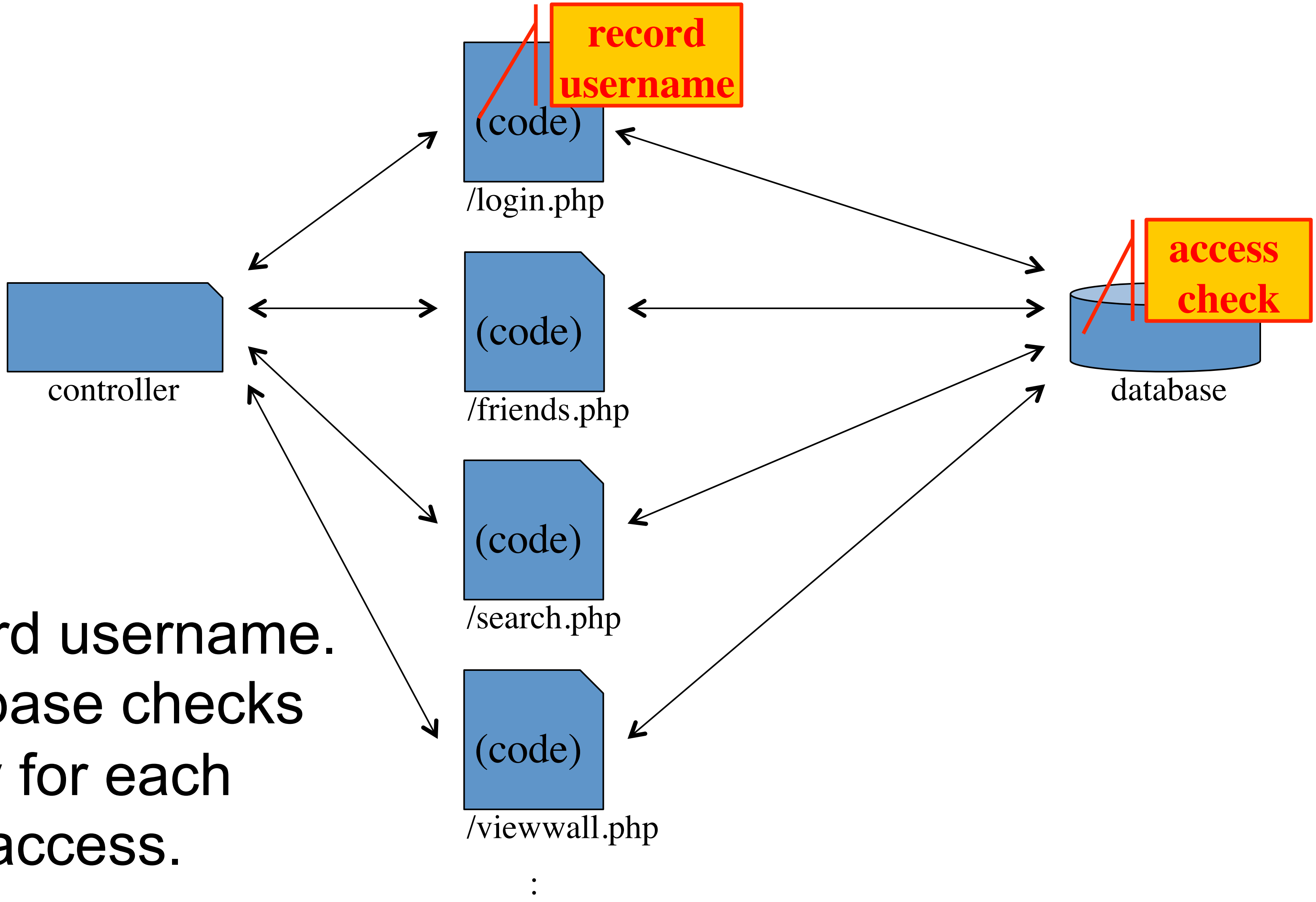
How should we implement access control policy?

Option 1: Integrated Access Control



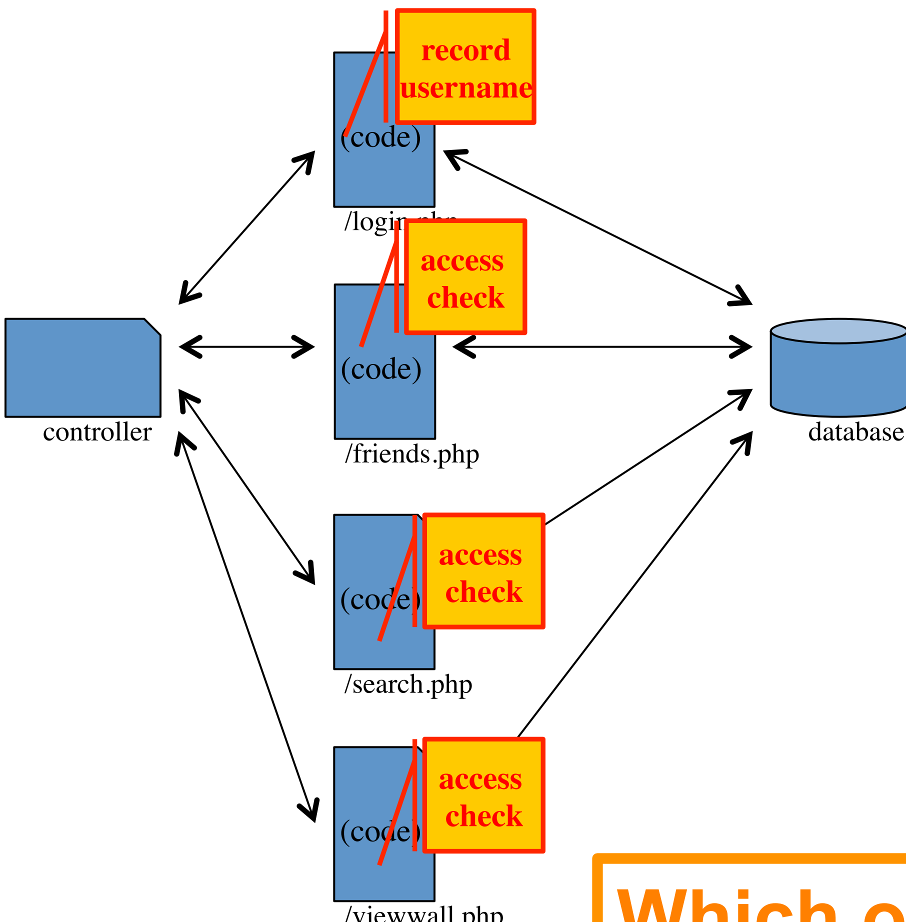
Record username.
Check policy at each place in code that accesses data.

Option 2: Centralized Enforcement



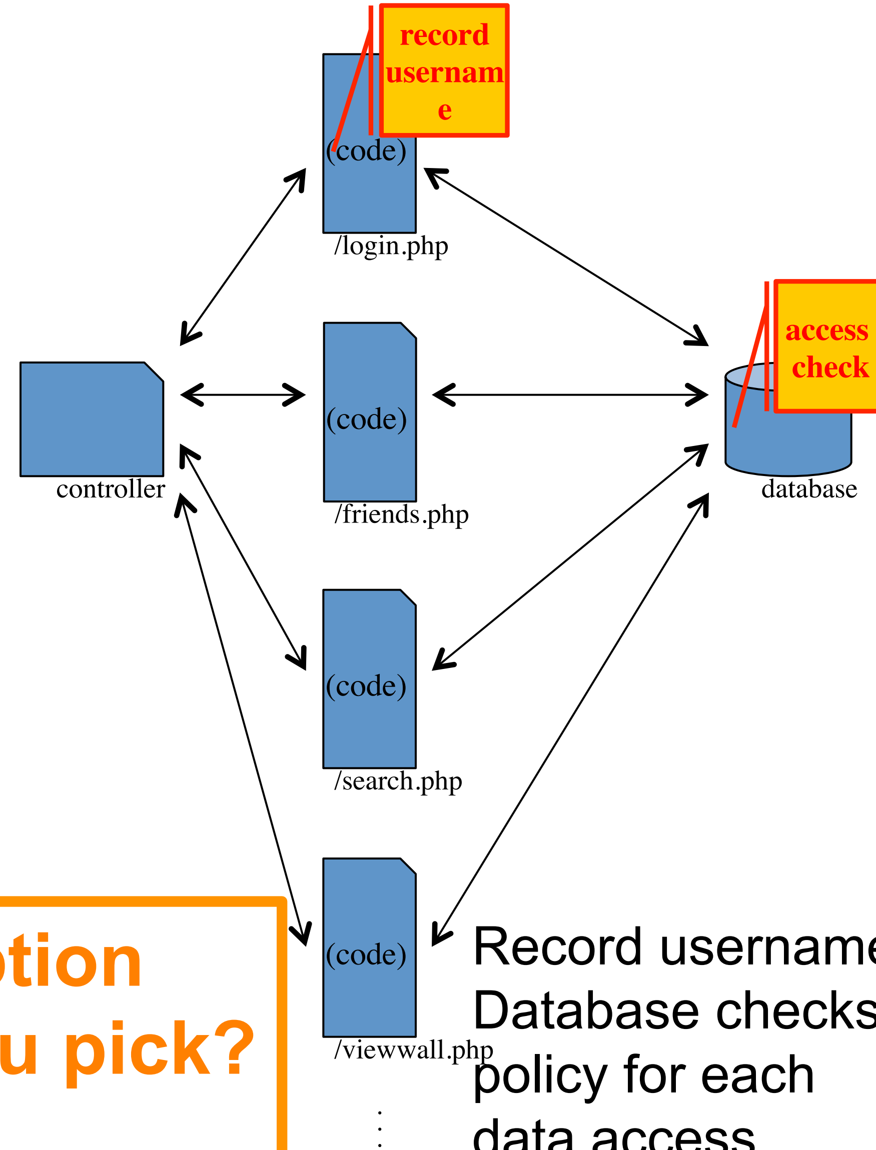
Record username.
Database checks policy for each data access.

Option 1: Integrated Access Control



Record username.
Check policy at each place in code that accesses data.

Option 2: Centralized Enforcement



Record username.
Database checks policy for each data access.

Which option would you pick? Discuss.

Analysis

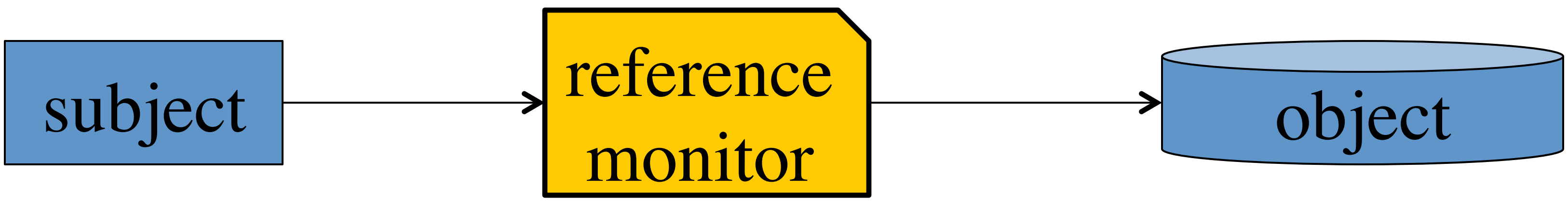
- Centralized enforcement might be less prone to error
  - All accesses are vectored through a central chokepoint, which checks access
  - If you have to add checks to each piece of code that accesses data, it's easy to forget a check (and app will work fine in normal usage, until someone tries to access something they shouldn't)
- Integrated checks might be more flexible

Complete mediation

- The principle: complete mediation
- Ensure that all access to data is mediated by something that checks access control policy.
  - In other words: the access checks can't be bypassed

Reference monitor

- A reference monitor is responsible for mediating all access to data



- Subject cannot access data directly; operations must go through the reference monitor, which checks whether they're OK

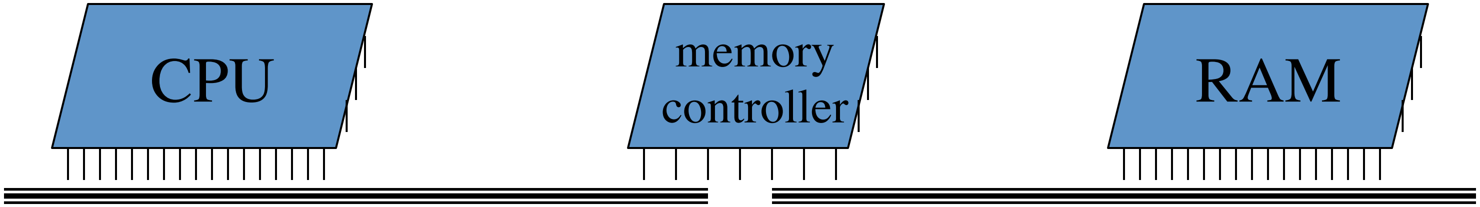
Criteria for a reference monitor

Ideally, a reference monitor should be:

- Unbypassable: all accesses go through the reference monitor
- Tamper-resistant: attacker cannot subvert or take control of the reference monitor (e.g., no code injection)
- Verifiable: reference monitor should be simple enough that it’s unlikely to have bugs

Example: OS memory protection

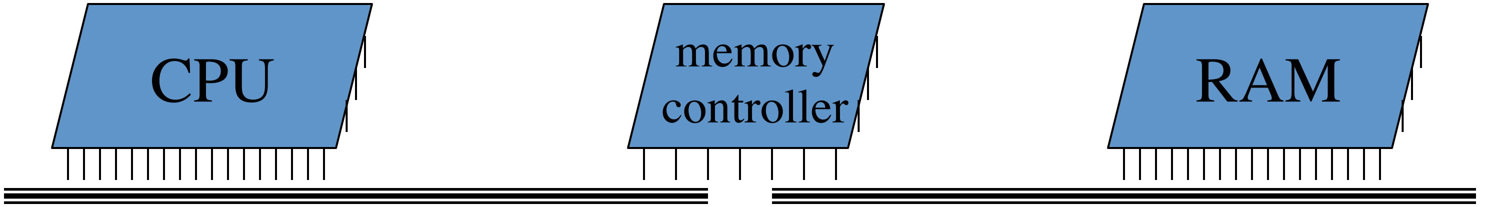
- All memory accesses are mediated by memory controller, which enforces limits on what memory each process can access



Unbypassable? ✔

Example: OS memory protection

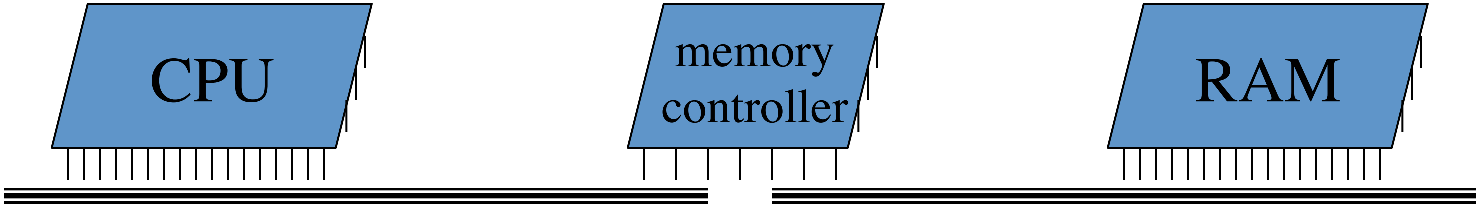
- All memory accesses are mediated by memory controller, which enforces limits on what memory each process can access



Tamper-resistant? ✔

Example: OS memory protection

- All memory accesses are mediated by memory controller, which enforces limits on what memory each process can access



Verifiable? ✔

TCB

- More broadly, the trusted computing base (TCB) is the subset of the system that has to be correct, for some security goal to be achieved
  - Example: the TCB for enforcing file access permissions includes the OS kernel and filesystem drivers
- Ideally, TCBs should be unbypassable, tamper-resistant, and verifiable

Coming Up …

- Homework 1 due Monday
- Buffer overrun review session, Thursday, 6-8pm, 155 Dwinelle